USENIX Association

Proceedings of BSDCon '03

September 8–12, 2003
San Mateo, CA, USA

Conference Organizers

Program Chair

Gregory Neil Shapiro, *Sendmail, Inc.*

Program Committee

Jason Evans, *Canonware*
Sam Leffler, *Errno Consulting*
Michael Lucas, *FreeBSD Documentation Project*
Kostas Magoutis, *Harvard University*
Todd C. Miller, *OpenBSD*
Donn M. Seeley, *Wind River Systems*
Bill Squier, *NetBSD*
Gregory Sutter, *Daemon News*
Robert Watson, *NAI Labs, FreeBSD Project*
Christos Zoulas, *NetBSD*

The USENIX Association Staff

BSDCon '03

September 8–12, 2003
San Mateo, CA, USA

Index of Authors v

Message from the Program Chair vii

Wednesday, September 10, 2003

Sticky Problems
Session Chair: Donn Seeley, Wind River Systems

Reasoning about SMP in FreeBSD 1
Jeffrey Hsu, FreeBSD Project

devd—A Device Configuration Daemon 7
M. Warner Losh, Timing Solutions, Inc.

ULE: A Modern Scheduler for FreeBSD 17
Jeff Roberson, The FreeBSD Project

Release Engineering
Session Chair: Gregory Sutter, Daemon News

An Automated Binary Security Update System for FreeBSD 29
Colin Percival, Computing Lab, Oxford University

Building a High-performance Computing Cluster Using FreeBSD 35
Brooks Davis, Michael AuYeung, Gary Green, and Craig Lee, The Aerospace Corporation

build.sh: Cross-building NetBSD 47
Luke Mewburn and Matthew Green, The NetBSD Foundation

Thursday, September 11, 2003

Storage/Crypto
Session Chair: Robert Watson, NAI Labs, FreeBSD Project

GBDE—GEOM Based Disk Encryption 57
Poul-Henning Kamp, The FreeBSD Project

Cryptographic Device Support for FreeBSD 69
Samuel J. Leffler, Errno Consulting

Enhancements to the Fast Filesystem to Support Multi-Terabyte Storage Systems 79
Marshall Kirk McKusick, Author and Consultant

System Building
Session Chair: Michael Lucas, FreeBSD Documentation Project

Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions 91
Hideki Eiraku and Yasushi Shinjo, University of Tsukuba

A Digital Preservation Network Appliance Based on OpenBSD 103
David S. H. Rosenthal, Stanford University Libraries

Using FreeBSD to Render Realtime Localized Audio and Video 115
John H. Baldwin, The Weather Channel

Friday, September 12, 2003

Networking
Session Chair: Kostas Magoutis, Harvard University

Tagging Data in the Network Stack: mbuf_tags 125
Angelos D. Keromytis, Columbia University

Fast IPSec: A High-Performance IPsec Implementation 133
Samuel J. Leffler, Errno Consulting

The WHBA Project: Experiences "deeply embedding" NetBSD 141
Jason R. Thorpe and Allen K. Briggs, Wasabi Systems, Inc.

Index of Authors

AuYeung, Michael 35
Baldwin, John H. 115
Briggs, Allen K. 141
Davis, Brooks 35
Eiraku, Hideki 91
Green, Gary 35
Green, Matthew 47
Hsu, Jeffrey 1
Kamp, Poul-Henning 57
Keromytis, Angelos D. 125
Lee, Craig 35
Leffler, Samuel J. 69, 133
Losh, M. Warner 7
McKusick, Marshall Kirk 79
Mewburn, Luke 47
Percival, Colin 29
Roberson, Jeff 17
Rosenthal, David S. H. 103
Shinjo, Yasushi 91
Thorpe, Jason R. 141

Message from the Program Chair

Welcome to BSDCon '03 and San Mateo, California. This is the fourth U.S. BSDCon conference and the second to be sponsored by the USENIX Association. Thanks to the efforts of many people, we've ended up with a great set of tutorials, papers, and invited talks. As in past years, the success of this year's conference proves the strength and continued growth of the BSD operating systems.

The refereed papers track is the result of the hard work of the program committee and, of course, all the authors who submitted their work for review. Each of the 35 submissions were read by at least three of the eleven members of the program committee. In the end, we accepted 15 papers for five technical sessions in the areas of solutions to sticky problems, system updating and release engineering, storage and cryptography, system building, and networking and embedded systems.

I would like to take this opportunity to thank the many people who helped make this conference a success. Clearly, this conference wouldn't exist without the dedication of the program committee, who recruited authors and invited talks speakers, and also reviewed submissions and shepherded refereed papers. In particular, Sam Leffler, last year's program chair, not only returned as as a member of the program committee, but also provided helpful guidance throughout the creation of this year's conference.

This conference would not have happened without the support of the USENIX staff and the USENIX Board of Directors. Dan Klein did his usual great work in organizing the tutorials in various kernel topics and advanced BSD security. Special thanks go to USENIX President Kirk McKusick, who has repeatedly championed BSDCon as a USENIX-sponsored activity, and to Ellie Young, Jane-Ellen Long, and Barbara Freel of the USENIX staff. Most important, I thank the authors, speakers, and participants for making this year's conference a success.

Gregory Neil Shapiro
Program Chair

Reasoning about SMP in FreeBSD

Jeffrey Hsu
The FreeBSD Project

Abstract

While the subject of SMP locking primitives has been well covered in the literature [Val][Schm][Bald][Leh], there has been relatively little discussion on the rationale and process behind the application of these locking primitives. This leads to an inverted problem where the bulk of the work in making a kernel SMP-safe lies above the locking primitives, yet there is little guidance on how and where to lock in the individual subsystems. In this paper, we will discuss our experiences with SMP locking in the FreeBSD kernel and illustrate some of the reasoning concerning the placement or non-placement of SMP locks in the kernel. We hope this will aid other developers in locking up the remaining subsystems and in understanding the locks that are already in place. We start with an overview of general locking strategies followed by many examples of race conditions caused by faulty SMP reasoning and give solutions for correctly locking up the affected piece of code. All our examples are taken from actual committed versions of the FreeBSD source.

1. Global Reasoning

Our first guideline pertains to the difference between global reasoning and local reasoning. To paraphrase a famous quote about distributed systems[Lamp], SMP locking is where some code that you didn't even know existed can break your own local code.

Guideline #1: Think globally.

For SMP to work properly, all the affected code must adhere to the same locking strategy. A single piece not under the necessary lock can render all the other locks useless. For example, one common race condition concerns read-modify-write operations. A race window exists between the time a processor does the read and before it does the write, whereby another processor does a write.

processor 1	processor 2
read	
modify	
	read
	modify
	write
write	← loses processor 2's write!

The most frequently used solution for this race is to place locks around all the read-modify-writes of this variable. However, missing a lock can mysteriously result in a write being lost. We see this race in rev modify-write operations on the f_flag field, but only one of which holds the mutex lock.

```
fcntl():
        ...
        case F_SETFL:
        ...
        fp->f_flag &= ~FNONBLOCK;

flock():
        ...
        FILE_LOCK(fp);
        fp->f_flag |= FHASLOCK;
        FILE_UNLOCK(fp);
```

Here, even though the flock() code takes care to perform its operation while holding the mutex lock for a file structure, a fcntl() operation could sneak in and modify some other, totally different, bit in the f_flag field, causing the result of the flock() to be lost!

fcntl	flock
read	
modify	
	lock
	read
	modify
	write
	unlock
write	← loses flock() result!

Furthermore, the result of either operation can be lost due to the missing lock, not just the place where the lock is missing, as is illustrated by the following sequence of operations:

fcntl	flock
	lock
	read
	modify
read	
modify	
write	
loses fcntl() result! →	write
	unlock

In this case, both operations occur in the same file in the well-examined kern/ directory, so this bug should have been easy to detect. However, there are uses and assignments of the f_flag field in far off places like dev/streams/streams.c which need to be examined for potential races.

2. Understand the underlying code

A closely related principle to knowing all the places where a field is used is to understand what the underlying code does. Only by understanding what the subsystem is trying to accomplish can a proper locking strategy be devised. Many cases of improper SMP locking can be traced back to a misunderstanding about what the underlying code does.

Guideline #2: Understand the code to be locked.

For example, in rev 1.79 of uipc_usrreq.c, file descriptor table locks were added in the following code in unp_attach():

```
FILEDESC_LOCK(curproc->p_fd);
unp->unp_rvnode = curthread->td_proc->p_fd>fd_rdir;
FILEDESC_UNLOCK(curproc->p_fd);
```

Figure 1

This example locks a single read statement of the word-sized fd_rdir field. A word-sized memory read operation is already an atomic operation [Schimmel. Neither the fd_rdir nor the unp_rvnode fields are accessed in the rest of the routine. As we shall see later on, placing locks around a single read statement as is done here is rarely the right thing to do. But more importantly, the field that is being updated, unp_rvnode, is obsolete and is not used anywhere else in the code! So, the correct thing to do here should have been simply to delete this line, rather than wrapping a useless assignment inside a mutual exclusion lock. Here is a case where knowing what the unp_rvnode field is used for or not used for would have resulted in leaving out the file descriptor table lock altogether. An application of the first guideline would have helped in discovering this.

A proper approach to SMP locking starts with asking what the underlying subsystem is trying to do, then asking which of these operations involve races, and finally, which type of locks are appropriate to close these races. This is in marked contrast to a bottoms-up approach of locking individual statements considered in isolation, which does not work.

3. Naive SMP Locking

While the first principle requiring locks to be globally applied would seem to dictate that everything should be wrapped inside locks, this is not the case. In fact, a naive "wrap all accesses inside locks" approach is rarely the right thing to do and leads to unnecessary locking where locks are not needed and missing locks where locks are needed. The places where this approach gets a lock right, it's by accident. So, this guideline pertains with what not to do.

Guideline #3: Don't simply wrap all accesses inside locks

We already saw a consequence of this in Figure 1, where file descriptor table locks were placed around a single atomic read of a field in the file descriptor table structure. Another example of an unnecessary lock is in rev 1.79 of uipc_usrreq.c,

```
FILEDESC_LOCK(td->td_proc->p_fd);
vattr.va_mode =
 (ACCESSPERMS & ~td->td_proc->p_fd-
>fd_cmask);
FILEDESC_UNLOCK(td->td_proc->p_fd);
```

which is an atomic read of the fd_cmask field with no other reads of any other file descriptor table structure fields nearby.

Corollary: Don't lock single atomic memory reads.

Consider the following generic structure,

```
struct something {
        int field;
```

```
        int otherfield;
} *p;
```

There are few cases where

```
LOCK(p)
x = p->field
UNLOCK(p)
```

can protect anything.

If someone else is writing to p->field, then that write either occurs before or after this read. Even with locks, both of the following two cases are possible.

processor 1	processor 2
LOCK(p)	
modify p->field	
UNLOCK(p)	
	LOCK(p)
	x = p->field
	UNLOCK(p)

or

processor 1	processor 2
LOCK(p)	
x = p->field;	
UNLOCK(p)	
	LOCK(p)
	modify p->field
	UNLOCK(p)

So the lock doesn't help determine an order here. The variable x could hold either the new or the old value of p->field. The read of this field might as well be unlocked. (The same argument applies to the store memory barrier effect of the LOCK() and UNLOCK() operations --- the store could occur either before or after the read, so it has no effect on determining the value read.)

But, one case where locks would be required is if the field temporarily holds a value that no one else is supposed to see and the writer, operating with the lock held, will store a valid value before releasing his lock. In this case, both the writer and reader need to hold the lock before accessing this field.

The situation is different if multiple fields in the structure were being read and we wanted to guarantee their read values were mutually consistent. Here, the reader would hold a lock around the multiple read statements and the writer would hold a lock across the multiple write statements. Then both readers and writers would always see a consistent picture of the fields. So, in the case, the correct course of action here is to expand the scope of the lock to cover all the nearby reads without releasing the lock in between. Then it is not a lock around a single read statement, but around multiple statements.

Finally, one might lock a structure in order to guarantee that it does not get freed while its fields are being used, but this is not the case for the examples mentioned. Locking to avoid deallocation during use usually requires some other lock to protect the initial call to LOCK(). Often, this problem is better solved with a reference count, a scheme that we will discuss next.

This section gives some valuable guidelines on what needs to be locked and why. Don't put a lock around everything just to put a lock around everything. As shown, that strategy can go wrong by having locks that cover too few statements, cover statements which don't need to be covered, and gives a false sense of security concerning the ordering between read and write operations. The goal in SMP locking is not to serialize everything through mutex locks, but to allow as much parallelism as possible while maintaining SMP safety.

4. Reference Counting Strategy

One of these SMP safety issues concerns guarding against deallocation. Reference counting is a scheme frequently used in SMP systems to protect against deallocation while an object is still in use. When a reference to an object is returned or stored, the reference count for that object is incremented. When a reference is no longer needed, the count is decremented, and if zero, the object is freed. The reference count manipulation must be performed under a mutual exclusion lock to avoid race conditions involving the count. These rules must be followed uniformly for the protection to work and to avoid memory leaks.

A disadvantage of the reference counting strategy is that a lock is typically acquired and released twice in the process of incrementing and decrementing the reference count. These count manipulation operations are an example of a read-modify-write operation mentioned earlier. (An alternative is to use the atomic operations in <sys/atomic.h>, but those are not applicable to reference counting scenarios where some set of actions must be performed atomically. They also incur the same amount of lock overhead at the machine level.)

5. Protect The Initial Reference

One of the problems encountered when implementing a reference counting scheme is how to protect the initial reference. Before locking an object to increment its reference count, we must ensure that object doesn't get freed before it is locked.

```
get pointer to object
                                ← obj freed here!
lock(ptr->mtx)
increment ptr->refcnt
unlock(ptr->mtx)
```

This problem is a particularly thorny one and can be addressed in the FreeBSD kernel in a number of ways, all of which involve reasoning inductively from some base case. For example, system calls occur within a process context, so system call code can safely lock the proc structure first before validating and acquiring a substructure lock. In the absence of such a natural base case, a global lock can be used in its place.

Another solution involves the transfer of a reference count lock protected by already holding an existing count or sole reference. For example, on initial allocation of a structure and before any references to that structure are made accessible, the allocation routine increments the reference count to 1. The calling routine can then store the reference in some other structure, taking care to increment the count as necessary. Any code that later accesses that reference knows by induction that a visible reference is a valid one.

6. Guard Against Deallocation

The deallocation during use problem looks like

```
processor 1                     processor 2
get obj
var1 = obj->field1
                                free(obj)
var2 = obj->field2  ←  use after free!
```

Holding the lock is one strategy to guard against deallocation during use.

```
processor 1                     processor 2
lock(obj)
        waits for unlock →  lock(obj)
uses of obj
unlock(obj) ← indicates obj no longer in use
                                free(obj)
```

The strategy of holding the lock to guard against deallocation during use must be used in conjuction with another strategy to guard the initial reference. The advantages of holding the lock to guard against deallocation is that it is simpler and lower overhead than reference counting, which requires a lock to atomically increment the reference count and another lock to atomically decrement it. It does, however, limit concurrency. With reference counting, simultaneous use of an object is allowed.

```
processor 1                     processor 2
lock obj
increment refcnt
unlock obj
                                lock obj
                                increment refcnt
                                unlock obj

use obj ← simultaneous use → use obj

lock obj
decrement refcnt
unlock obj
                                lock obj
                                decrement refcnt
                                unlock obj
```

An example of a place where mutexes are used to guard against deallocation is the inpcb lock in the networking stack. Here, due to the serial nature of the operations performed while holding an inpcb lock, only one processor can access the structure at a time anyways, so holding the lock to guard against deallocation rather than obtaining a reference count does not artificially limit parallelism.

7. Protect Linked Lists

Linked list traversal and linked list manipulation must be performed under a lock. This lock must be common to all the readers and writers of this linked list. One example of faulty locking is in rev 1.90 of sys_pipe.c, which uses the pipe lock in many places to protect knote list traversal, for example, in filt_pipedetach():

```
PIPE_LOCK(cpipe);
SLIST_REMOVE(&cpipe->pipe_sel.si_note,
                kn, knote, kn_selnext);
PIPE_UNLOCK(cpipe);
```

Unfortunately, the knote() function in kern_event.c, which walks the knote list, is called without holding the

pipe lock, leading to a race condition between the two operations. This bug is an example of not holding the same SMP lock for list traversal and list manipulation. This can be addressed, after tracing the calls of the knote() function back to pipeselwakeup(), by ensuring that pipeselwakeup() is always called with the pipe lock held. Alternatively, a new lock can be introduced to protect the knote list and acquired during both list traversal and list manipulation.

If a list will be traversed by more than one thread simultaneously, it may pay to use a shared sx lock [Bald], as is done for the some of the process lists. From the pfind() routine in kern_proc.c,

```
sx_slock(&allproc_lock);
LIST_FOREACH(p, PIDHASH(pid), p_hash)
        ...
sx_sunlock(&allproc_lock);
```

Here, the allproc_lock shared lock is used to protect the linked list pointer stored in the p_hash field. This is a shared lock, so multiple threads can be executing this code in pfind() simultaneously. When modifying this field, an exclusive lock must be obtained to lock out other readers and writers, as is done, for example, in exit1() in kern_exit.c:

```
sx_xlock(&allproc_lock);
LIST_REMOVE(p, p_hash);
sx_xunlock(&allproc_lock);
```

Unfortunately, shared locks are many times more expensive than simple mutexes, so they should only be used when lock profiling indicates lock contention for a given lock. Since most locks are not contested, using a shared lock rarely pay off.

One technique used in the FreeBSD code to leave open the option to switch from a fast simple mutex to a slow shared sx lock is to use macro definitions for the locking. We see this in net/if_var.h,

```
#define IFNET_WLOCK()    mtx_lock(&ifnet_lock)
#define IFNET_WUNLOCK()
                         mtx_unlock(&ifnet_lock)
#define IFNET_RLOCK()    IFNET_WLOCK()
#define IFNET_RUNLOCK() IFNET_WUNLOCK()
```

where shared read locks are differentiated from exclusive write locks in the code, but the two types of lock usage are both defined as the same simple mutex lock. This allows for an easy switch to a shared sx lock if lock profiling later determines that this is a heavily contested lock and the code usage exhibits a strong multiple-readers single-writer pattern. The amount of lock contention will depend on the application mix being run as well as the number of processors in the system.

8. Lock-Free Synchronization

There are many opportunities to exploit lock-free synchronization techniques in an SMP setting. The FreeBSD kernel does not, as yet, use many of these techniques. One technique that is currently deployed is the use of a generation count on a structure. This is a count that is incremented each time a structure is modified or freed. The generation count is remembered before an unlocked operation and checked at the end of an operation. If the generation count hasn't changed, then the structure was not modified during the operation. For example, this technique is used to avoid locking structures that are copied out to user-land for sysctls.

9. Lock Ordering and Deadlocks

Lock ordering considerations pervade much of the locking in the FreeBSD kernel.

There are four necessary conditions [Silb] for deadlock:

1. mutual exclusion
2. hold and wait
3. no preemption
4. circular wait

Breaking any one of these conditions is sufficient to guarantee that no deadlock can occur. The most commonly used approach and the one FreeBSD has chosen by design is to order the locks.

This means that locks are acquired in a particular order and if a lock is required while a higher numbered lock is held, the higher numbered lock is released and lock acquisition proceeds anew from the top. The witness facility automatically tracks lock ordering and warns if it detects locks being acquired out of order.

10. The dreaded "Could sleep while holding lock" warning

Blocking memory allocations are detected by witness and commonly reported by FreeBSD-current users. There are several common strategies for dealing with this. One is to allocate before acquiring mutex. Another is to allocate after releasing mutex. Sometimes a block-

ing malloc can be avoided by copying into local variables. Finally, if all else fails, use non-blocking allocation.

11. Related Work

There has been much work in the past concerning the formal semantics of programs. Some of this targets formal reasoning about concurrent programs [Lamp2]. In general, with notable exceptions such as [Sav], few of these techniques have not been applied to programs as large as the FreeBSD operating system.

12. Summary and Future Work

Many of the subsystems still remain to be locked up. An analysis of what the subsystem does, the inherent race conditions, and the proper locking strategy should precede placing actual locks in the code. That all the bugs discussed here were found in committed code indicates that we should focus on SMP safety and completeness. After that has been accomplished, then we can turn out attention towards lock profiling, tuning, and reorganizing code and data structures to better take advantage of the SMP environment.

We have gone over some of the techniques used in locking up the subsystems in the FreeBSD kernel. Both examples of correct as well as incorrect SMP locking and the reasoning behind both were explored. The common question of what needs to be locked and what does not need to be locked was illustrated in a number of contexts within the kernel. We hope this aids developers in understanding the current locking employed in the FreeBSD SMP kernel and in locking up the remaining modules.

13. References

[Bald] John Baldwin, Locking in the Multithreaded FreeBSD Kernel, Proceedings of the BSDCon 2002 Conference, Usenix, 2002.

[Lamp] Leslie Lamport, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." CACM, June 1992.

[Lamp2] Leslie Lamport, "A Temporal Logic of Action", ACM Transactions on Programming Languages and Systems, May 1994.

[Leh] Greg Lehey, Improving the FreeBSD SMP Implementation, Proceedings of FREENIX Track: 2001 USENIX Annual Technical Conference, Usenix, 2001.

[Sav] Stefan Savage, Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs, ACM Transactions on Computer Systems, Vol 15, No 4, Nov 1997.

[Schm] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers, Addison-Wesley, 1994.

[Silb] Silberschatz, Operating System Concepts, 6th Edition, John Wiley & Sons, 2001.

[Val] Vahalia, UNIX Internals: The New Frontiers, Prentice Hall, 1995.

devd – A device configuration daemon

M. Warner Losh
Timing Solutions, Inc
Boulder, Co
imp@bsdimp.com

Abstract

Hot–pluggable bus technologies have proliferated, rendering traditional boot time configuration of devices via an `/etc/rc` script insufficient for many user's needs. Most implementations of hot–plug technologies have provided a means to address these deficiencies, yet their solutions tend to be confined to only that technology. The goal of FreeBSD's devd(8) is to provide a uniform framework by which interesting events relating to hot–plugging can be handled. devd provides a regular framework for these technologies to have user–land configuration commands run in a generic, extensible way. The implementation encountered a number of issues which are instructive to explore. devd only responds to events that the kernel generates and does not participate in interactions with the kernel that would block another thread of execution. At the present time, devd supports executing arbitrary commands when a driver attaches to the tree, when it detaches and when a bus detects an unknown device attached to that bus.

1 Problem Statement

When Unix was originally designed, all the devices that the machine had were hardwired into the kernel. Over time mechanisms evolved to allow users to configure which devices their kernels should have. As technology improved, hardware became hot–pluggable and self identifying. Traditional Unix kernels are awkward to use on a machine where devices dynamically change after the kernel has booted. A number of hot–plug technologies have been brought to market over the past few years: 16–bit and 32–bit PC Card[PC Card], IEEE 1394–1995[Firewire], HotPlug PCI[PCIHotPlug], USB 1.0 and 2.0[USB], SCSI, ATA, SATA, and others.

Traditional Unix systems could only configure network interfaces, or otherwise bring devices into a useful state during the boot process through `/etc/rc` or similar mechanisms. Since many hot–pluggable devices do not exist at boot, some other mechanism was needed to bring these devices to a useful state.

Typically, hot–plug technologies have been integrated in an ah–hoc way. For example, the PC Card daemon `pccardd` in FreeBSD handles only 16–bit PC Cards. The USB daemon `usbd` in FreeBSD and NetBSD handles only USB devices. Each of these daemons does its job well, but is specialized to its particular technology. These daemons have low reusability for new technologies. Each new technology needs a new daemon to be written. There's presently no daemon to handle firewire devices on either FreeBSD or NetBSD, for example. The proliferation of daemons takes up additional resources, and wastes programming effort for each new technology.

A related problem is that sometimes devices are attached to busses for which no driver exists in the kernel. Bus technologies that support hot–plugging of devices typically have some sort of so called "Plug and Play" identifiers that can be used to intelligently select the driver. In FreeBSD there is only limited support for loading device drivers when unknown devices are present. Only `usbd` supports loading drivers when unknown hardware is plugged in, and nothing supports augmenting drivers for unknown devices encountered during the boot phase.

Many of these busses also provide "location" information to uniquely identify multiple instances of the same kind of card. While not necessarily used by most devd users, this location information is exposed in a uniform way. This uniformity allows users to configure their system based on where a device is, rather than what order it happens to probe

in. Prior to the devd efforts, FreeBSD could only find the location of a device for most busses via bus specific methods that often proved unreliable.

devd was conceived to try to solve these problems. devd will be used in this paper to refer to all parts of the system, even though the devd daemon is only part of the solution.

2 Prior Art

A number of ad–hoc solutions to these problems have existed over the years. Each one of them solved a small slice of the problem set. My search for prior art didn't uncover any truly generic method for all drivers in a system.

2.1 FreeBSD pccardd

The original PC Card implementation in FreeBSD, sometimes called OLDCARD, has a split user-land kernel implementation. The bare minimum of functions that were required in the kernel were implemented there, but the bulk of the PC Card configuration procedures were implemented in a daemon called `pccardd`. The kernel would tell this daemon of card insertions. The daemon would then parse the CIS information from the card, chose a driver and its resources based on a configuration file and tell the kernel which device to attach. When that was done, it would optionally run additional commands. It did similar things when a card was ejected. A companion program pccardc offered some additional control to the system, such as dumping a CIS for debugging purposes, or powering off a card.

This system worked fairly well when it was written. In those days, most of the hardware was still ISA based and everything was hardwired at a particular address. As systems became more complicated, problems arose with resource conflicts. In addition, as bus technologies changed from edge triggered interrupts to level triggered interrupts, the split between user–land and the kernel was found to be suboptimal because some of the things that the user–land daemon was doing would cause interrupt storms because the kernel didn't expect to cooperate with `pccardd` for those operations.

2.2 NetBSD usbd

NetBSD's `usbd` provided a way for a user-land process to react to events in the usb system and to force a traversal of the usb tree. NetBSD moved this functionality into a kernel thread. FreeBSD expanded its `usbd` similarly to FreeBSD's `pccardd` to allow for aribtrary commands to be run on attach and detach. The underlying usb driver provided hooks for additional events that were never used by `usbd`. In addition, all the usb configuration data comes from the kernel, rather than from parsing the data like `pccardd` does.

2.3 Windows

While Windows doesn't exactly support running a command when a device attaches or detaches from the device tree, it does define a similar C API. Windows solves the problem of which driver to load and attach differently. In the windows world, a configuration file associates a driver to a plug and play ID and the device manager, not the device driver, decides which devices a driver services.

2.4 MAC OS/X

OS/X facilities are similar to Windows. OS/X uses XML based config files to determine which device driver is attached to a given device. I am unaware of command execution when a driver attaches or detaches.

3 Design Overview

3.1 Unix Philosophy

The Unix philosophy has been to keep things as simple as possible. Each tool does one small job, and does it every well. Complex problems are solved by stringing together simple tools that solve orthogonal problems. Much like cat(1) does not try to implement paging functionality, devd does not try to dictate policy to the kernel. devd is reactionary: it only responds to events that have already happened. It

does not participate in the decision making process of kernel device configuration.

3.2 Big Picture

Let's look at a typical case to illustrate how the process works. A wireless card is inserted into a CardBus slot. The CardBus bridge notices this has happened, and causes all drivers with CardBus attachments to be probed. One of them claims the card, and then its attach routine is called. After a successful attach, an event is sent to the devd daemon. The daemon inspects the event and decides that it is configured to start `dhclient` and does so.

All the data flow is one way. Some event happens, which causes a device driver to attach. That act causes an event to be sent to devd, which does something. devd only reacts to events that have happened in the past. devd does not participate in the bidding on a device. The kernel will never block waiting for data from devd.

3.3 Kernel Driver

FreeBSD stores the kernel device driver tree in a generic format for all devices in the system. Since manipulation of the tree is confined to a few well known routines, the first part of devd is a driver that hooks into this code. This driver queues this information and delivers it to user–land via the `/dev/devctl` device. Each message is one line of text. The messages have a well defined meta-format so messages not known to a particular client may easily be discarded without loss of synchronization in the protocol stream.

3.4 User–land Daemon

The user–land daemon, called devd, processes the events from the kernel and dispatches them based on devd's configuration file `/etc/devd.conf`. This dispatching is regular expression based. Any information that the kernel provides about the event can be matched. Actions are arbitrary Unix commands.

4 Implementation Details

4.1 Kernel portion

The kernel portion of devd is implemented in `/usr/src/sys/kern/subr_bus.c`. Hooks in the attach, detach and nomatch portions of the NEWBUS code queue events, if `devctl` is not disabled. The queue depth is unlimited, but it is cleared when `devctl` is disabled early in the boot process for those systems that choose not to use devd's features. Look for the functions `devadded()`, `devremoved()` and `devnomatch()` for details. Since `devctl` hooks in at such a low level of FreeBSD's NEWBUS system, all busses in the system are automatically supported at a rudimentary level.

The interface to the kernel is through the `/dev/devctl` device driver. The devd daemon opens this device, and reads events from the queue using the standard read(2) calls. Each event is on a line by itself, in ascii format. In theory, this interface would support a devd-like program written in Perl, Ruby or some other programming language. `devctl` also supports ioctls for attaching and detaching devices, but those ioctls are beyond the scope of this paper. `devctl` presently supports only one reader, but future versions will have a cloning driver that will allow multiple readers as people write status programs that wish to listen to the event stream from the kernel.

4.2 Bus driver

While full support of all the devd functionality requires bus drivers to implement several functions, minimal support for devd only requires bug free operation of the underlying bus.

The most basic requirement is that busses must provide NEWBUS attachments to their children. Busses may provide additional information about location and identifying characteristics of the device. Finally, they may also support bus rescanning when a new driver which has attachments on that bus is loaded.

Most of the busses in the system do provide NEWBUS attachments for each of their children. Since it mostly predates the NEWBUS system, CAM

doesn't use NEWBUS children for the SCSI and ATAPI devices it finds. Since the usb stack was ported from NetBSD, not all of its devices are NEWBUS devices. For these busses, the support for them from devd is limited. All the other busses in the tree that have been examined use full NEWBUS children for all the devices attached to the bus.

While all the busses using NEWBUS for their children are automatically supported at a basic level due to devctl's hooks inside of newcard, each bus can enhance its support for devd in three ways. First, it can implement a location kobj function, bus_child_location_str(), which devctl uses to tell devd where a given driver has attached to the bus. Second, the bus can report identification information with the bus_child_pnpinfo_str() method. Finally, to support automatic loading of drivers for unknown devices, the bus needs to support rescanning unattached devices when a new driver is loaded.

Location information is a bus specific way of locating the device, possibly uniquely. Many busses have a notion of a slot number where the card with the function(s) resides. Other busses have a notion of where the device lives within a tree of objects. The information returned by the bus_child_location_str, method can therefore be used by the user locate definitively the device.

Information about the nature of the device may also be supplied. This information is similar to the plug and play strings that Windows generates and uses in its device configuration. Busses wishing to implement this are required to implement the bus_child_pnpinfo_str kobj method. This method generates strings which are presented to devd by way of devctl. These string aren't strictly required, as devctl will simply pass less information about the device up to devd when they aren't present. When they are present, they can be used to select the driver to load that supports the hardware.

Finally, bus drivers that wish to fully support devd's functionality must also implement effective reprobing on module loading. NEWBUS provides hooks necessary to implement this feature, and most busses already support it. However, some busses attach a generic device to those devices that do not otherwise match. In those cases, the bus will need to detach that device driver from those children that are generically matched in order to allow the newly loaded driver a chance to bid on the device.

4.3 devd

devd dispatches commands based how a kernel event matches information from a configuration file. devd reads this configuration file on startup. This config file may contain a list of directories from which to read additional files. These additional files may contain anything that's in the main directory, and are only scanned once at startup. Directories are only scanned once, even if they are listed multiple times.

After devd reads it config file, it begins to process events. Since devd is a daemon, it will place itself into the background. There are two modes of operation for this. The first mode, the default, waits until all the events in the queue have been read and their actions dispatched. This is used during boot to eliminate a race later in the boot process between devd configuring a device and rc scripts using those results. The second mode of operation causes devd to become a daemon immediately after reading its config file. This mode can be used to optimize boot time when no such dependency exists.

There are three kinds of events that devctl produces for devd to consume. When a bus driver attaches a driver to a device on that bus, an attach event is generated. When a driver detaches from a device, the detach event is generated. When a bus driver detects a device on the bus no driver claims, a nomatch event is generated.

Attach and detach events happen after the fact. The driver is already attached or detached before the events happen. The command(s) that are run in a matching action must take this into account. This especially means that it is not possible to run any commands before the driver detaches. In controlled situations, this may be undesirable behavior, but when a card is ejected, say, from a PC Card slot that hardware is gone and there is no control possible. In addition, a fundamental design decision of devd was to have it be reactionary only. Having the kernel stall on devd before detaching the device was deemed undesirable. Two way communication is much harder than one way communication.

A nomatch event is generated when a driver cannot be matched to a device. This may happen if the driver for that device is not in the kernel, or if no such driver exists. The typical response for a nomatch event is to either load a driver (in the former

case), or to ignore the event (in the latter). Again, devd is reactionary. It does not participate in the bidding process for the device. However, FreeBSD's NEWBUS system allows devd to be reactionary, while still allowing the device to ultimately attach. When the driver is loaded, the bus driver(s) for that driver will rescan all of their unattached children. Any new driver(s) to the system will be eligible to attach to the unknown device on this rescan. Another implication of this is that if a few different drivers could attach to the device, all can be loaded and the right one will win the bidding. While the "Plug and Play" information is typically sufficient to select only a unique driver to load, in some rare cases it only selectes one of two drivers.

5 Configuring devd

All of the details about devd's config file can be found in the devd.conf(5) manual page on FreeBSD[devd.conf]. Briefly, the config file is similar to configuration files for such software packages as bind and dhcpd. A number of different sections are used to control devd's behavior. A section exists for global configuration information. In addition a number of sections control the actions to take on certain events. Finally, a number of different comment styles are provided.

5.1 Options Section

The options section contains different options for devd's operation. Configuration options are specified as the option name, followed by one or more words or strings as appropriate for the option. While many options exist, only two will be discussed here.

The **directory** option takes one string. This string specifies a directory to read additional files from. All the files in the listed directory are merged into the configuration of devd, as if their contents had been appended to the original devd.conf file. The directory option may be specified many times; however devd only scans any given directory once. This slightly arcane mechanism was designed to allow for packages to participate in devd with a simple addition of a file on install and a simple unlink on removal.

The **set** option takes two parameters. The first parameter is the name of the variable to set. The second variable is the string to set it to. devd sets a number of its own variables during event processing, discussed below, and the set command augments those variables.

5.2 Attach, Detach and Nomatch Section

Each of these sections is given a weight. For each action type, the sections are sorted in decreasing order. Each section consists of zero or more matching directives, and zero or more actions. When processing an event, the sections are scanned in their sorted order. The first one whose matching directives all match is considered the best match. Only the section with the best match has its actions executed. All actions will be executed, reguardless of their exit code.

Each **match** directive contains two strings. The first string is the keyword to match against. The second string is the regular expression to match against. Both of these strings have devd variables expanded before their use. Each device event contains a number of key value pairs that are the plug and play information and location information the parent device provides. This is discussed in the section on event generation above.

`device-name "foo"` is a shorthand for `match "device-name" "foo"`. It is nothing more than syntactic sugar.

The **action** directives have one string. This is the string to execute. Like the **match** directive, it too has its devd variables expanded before the strings are used.

All three types of sections have identical syntax. However, the **attach** section is run only on attach events from devctl; the **detach** section on detach events; and the **nomatch** section on nomatch events.

6 A Few Examples

A few examples will illustrate how devd works from end to end. Each section will contain an excerpt from a configuration file, a description of the problem and solution and a walk through of all the events that happen in the system.

6.1 Using a Wireless CardBus Card

In this example, we have just purchased a brand new Atheros wireless card, supported by the `ath` driver. Assume that the `ath` driver is compiled into the kernel. Let us further suppose that if this machine has a atheros card on a pci bus, we want some other mechanism to configure the card. Figure 1 shows the relevant portion of the devd configuration file for this example.

```
attach 10 {
match "bus" "cardbus[0-9]+";
device-name "ath[0-9]+";
action "/etc/wlan $device-name start"
};
detach 10 {
match "bus" "cardbus[0-9]+";
device-name "ath[0-9]+";
action "/etc/wlan $device-name stop"
};
```

Figure 1: Simple wireless configuration file.

In this example, the following events happen:

1. The user inserts an Atheros wireless card into one of the CardBus slots
2. The CardBus bridge notices the card has been inserted
3. The CardBus bus attaches the ath driver to the card
4. /dev/devctl generates an attach event similar to the following:

   ```
   +ath0 at slot=0 function=0 on cardbus1
   ```

5. devd reads this event
6. devd sets "device-name" equal to "ath0", "slot" to "0", "function" to "0" and "bus" to "cardbus1".
7. determines that the above attach clause is the best match
8. devd expands the strings and executes "/etc/wlan ath0 start"
9. start-wlan-card configures the ath0 interface and the user goes wireless
10. time passes
11. the user ejects the atheros card
12. the CardBus bridge notices that the card is gone and the ath0 driver detaches from cardbus0
13. /dev/devctl generates an attach event similar to the following:

    ```
    -ath0 at slot=0 function=0 on cardbus1
    ```

14. devd reads the event, set variables and matches the above detach clause.
15. devd executes "/etc/wlan ath0 stop"

6.2 Automatic Driver Loading

In this example, a driver for the ALi M7101 Power Management Controller called `apmc`. In this example, the system is booting an encounters an unknown device. Figure 2 shows the relevant portion of the devd configuration file for this example.

```
attach 10 {
match "bus" "pci[0-9]+";
device-name "apmc[0-9]+";
action "/etc/powermon $device-name start"
};
detach 10 {
match "bus" "pci[0-9]+";
device-name "apmc[0-9]+";
action "/etc/powermon $device-name stop"
};
nomatch 10 {
match "bus" "pci[0-9]+";
match "vendor" "0x10b9";
match "device" "0x7101";
action "kldload apmc"
};
```

Figure 2: Unknown device configuration file.

In this example, the following events happen:

1. The system boots and pci bus 2 is scanning its children
2. No driver accepts the device in slot 17
3. /dev/devctl generates a nomatch event similar to the following (line breaks have been inserted for clarity):

```
? vendor=0x10b9 device=0x7101
  subvendor=0x1265 subdevice=0x7101
  class=0x068000 at slot=17
  function=0 on pci2
```

4. eventually devd starts and reads events
5. devd reads the nomatch event, sets the variables, and picks the nomatch clause above.
6. devd executes "kldload apmc"
7. The pci bus rescans its unattached children because a new pci driver has been loaded
8. the `apmc` driver attaches to the device in slot 17
9. /dev/devctl generates an attach event similar to the following:

```
+apmc0 at slot=17 function=0 on pci2
```

10. devd reads this event, sets the variables and executes the event.
11. devd expands strings and executes "/etc/powermon apmc0 start"

7 Lessons Learned

A number of problems were encountered during the development and deployment of devd. This section details those lessons, as well as the solutions that were arrived at for them.

7.1 To queue or not to queue

The initial implementation of devctl didn't queue events if there was no daemon running. There was a problem with this. Devd would have then ignored removable devices present at boot. In this case the rc system could take over, but it is optimized to static devices, so some functionality that devd offers would be lost. In addition, devd would have seen detach events when/if the card was removed without a corresponding insertion event.

The solution was to always queue the events. This allowed devd to receive the attach events for everything in the system and to process them for each device. It also allowed devd load drivers for unknown devices. There was a problem with this solution also. It assumes that devd will run relatively early in the boot process to reclaim the memory used by `devctl`'s queue. However, the user can choose not to run devd, or devd could have a bug and exit. In these cases, unbounded event queue would consume kernel resources indefinitely.

The solution to the problems introduced by the first solution was to add a sysctl that would enable or disable devctl's queuing of events. This sysctl would also free any events in the queue, freeing their memory. This solution works well for most cases, and does exactly what it says it does. It was a simple matter to wire this sysctl to a startup script parameter to turn devctl off in the kernel. Since devd was disabled by default, this solved the resource utilization issue.

One problem encountered quickly during prototyping with this solution was that users thought devd was broken. The solution was to have devd automatically re-enable `devctl` when it was disabled on startup. This solved the foot shooting issue without introducing more problems.

From the original queueing question, I found and corrected four issues. It makes one think of the rabbits in Australia.

7.2 Ordering

The next problem with devd was one of ordering. I naively thought that running devd as early as possible would be the best possible thing to do. The thinking was that devd could be used to configure any sort of device, and one wants to do that as early in the boot process as possible. However, there were problems with this. Since the environment that exists early in the boot process is severely limited, the file systems that devd can access to configure the devices is limited to those files in the root filesystem. This environment is a fairly restricted one.

Many users require more features to configure their devices. Many scripts use languages in /usr/local, such as Perl or Ruby. A number of desirable binaries are in /usr/bin or /usr/sbin, such as awk or wicontrol. Placing things last is too late, since it precludes configuring network cards early enough to be used by daemons, etc. In the end, starting devd just before the rc.d scripts mounted the remote file systems. In this way, devd users could maximize its use of the local resources available, while at the same time maximizing the amount of the startup process that can utilize the dynamically configured devices.

Since both devd and /etc/rc.d are both used to configure the kernel devices from user-land, some conflicts arose. This configuration typically is forming ifconfig commands for network devices, mounting remote file systems, loading firmware in some devices and so forth. Since devd treats all devices the same, both dynamic and static, some method for making sure that devices weren't configured multiple times was needed. This issue was resolved by using the 'legacy' configuration methods first. The devd scripts have checks to make sure that devices aren't already configured before doing its configuration.

7.3 Racing in the background

In the prototype versions, devd became a daemon very early in the process, so as to not hold up the boot while it read and processed its config file. There was a subtle problem with this approach that took a while to understand. If there were dependencies on devd having configured a device in the rest of the startup code, devd might not have configured it by the time it was needed because devd was running in the background, asynchronously to the rest of the boot process. For example, if a network interface were configured and then an NFS filesystem mounted over that interface, devd may or may not be able to configure the card's network address before the nfs mount started attempted. This lead to a mount failure. If this configuration is done with soemthing like `dhclient`, the configuration time is variable and possibly significant. To address this problem, devd processes all of the outstanding events from `/dev/devctl` before becomes a daemon. This stalls the boot process until all the devices are configured, eliminating the race. Since some users do not desire this stalling, devd can also optionally become a daemon early in the process to not stall the rest of the boot process.

8 Acknowledgments

I would like to thank Poul–Henning Kamp, John Baldwin, Peter Wemm and Scott Long for reviewing early designs and providing valuable insight into the problems.

9 Future Work

devd is a good start. However, it is a very simple tool. It provides the basic configuration stuff. Additional information is necessary to keep the full state of a system. This is needed for those people that only want to run sshd, for example, when they have a network card installed. FreeBSD's driver configuration is in such a state that it is not easy to produce tables of "Plug and Play" information that easily could translate into devd config files for dynamic loading of device drivers from a minimal kernel.

Power management events would offer a good future expansion of devd's event structure. Suspend and resume events are currently handled by `apmd` for APM. However, ACPI is replacing APM and it has no equivalent to `apmd`. Having yet another daemon to process events seems wasteful, so it may be profitable to explore expanding devd to encompass these events. Network interface events, such as carrier detect or carrier loss, may also be fertile ground for future expansion. It is not clear if a separate daemon is needed for the network events, or if programs like `dhclient` could handle those events in a smarter manner.

10 Availability

The software described in this paper have been integrated into FreeBSD 5.1–RELEASE. FreeBSD is available free of charge for download from **http://www.freebsd.org/** in many different forms. Improvements to devd will be incorporated into future versions of FreeBSD.

References

[devd.conf] **/usr/share/man/man5/devd.conf.5**

[Firewire] **http://developer.apple.com/firewire/platform.html**

[FreeBSD] **http://www.freebsd.org/**

[Linux–HotPlug] **http://sourceforge.net/projects/linux–hotplug/**

[NetBSD] **http://www.netbsd.org/**

[OpenBSD] **http://www.openbsd.org/**

[PC Card] *PC Card Standard, Release 7.0*, PCMCIA, (February 1999). **http://www.pcmcia.org**

[PCIHotPlug] *PCI Hot–Plug Specification, Revision 1.0*, PCI Sig, (October 6, 1997). **http://www.pcisig.com**

[USB] **http://www.usb.org/developers/docs.html**

ULE: A Modern Scheduler For FreeBSD

Jeff Roberson

The FreeBSD Project
jeff@FreeBSD.org

Abstract

The existing thread scheduler in FreeBSD was well suited towards the computing environment that it was developed in. As the priorities and hardware targets of the project have changed, new features and scheduling properties were required. This paper presents ULE, a scheduler that is designed with modern hardware and requirements in mind. Prior to discussing ULE, the designs of several other schedulers are presented to provide some context for comparison. A simple scheduler profiling tool is also discussed, the results of which provide a basis for making simple comparisons between important aspects of several schedulers.

1 Motivation

The FreeBSD project began a considerable shift in kernel architecture for the 5.0 release. The new architecture is geared towards SMP (Symmetric Multi-Processing) scalability along with low interrupt latency and a real-time-like preemptable kernel. This new architecture presents opportunities and challenges for intelligent management of processor resources in both SMP and UP (Uni-Processor) systems.

The current FreeBSD scheduler has its roots in the 4.3BSD scheduler. It has excellent interactive performance and efficient algorithms with small loads. It does not, however, take full advantage of multiple CPUs. It has no support for processor affinity or binding. It also has no mechanism for distinguishing between CPUs of varying capability, which is important for SMT (Symmetric Multi-Threading).

The core scheduling algorithms are also O(n) with the number of processes in the system. This is undesirable in environments with very large numbers of processes or where deterministic run time is important.

These factors were sufficient motivation to rewrite the core scheduling algorithms. While the new features are important, the basic interactive and 'nice' behavior of the old scheduler were required to be met as closely as possible. In conjunction with this project, the scheduler was abstracted and placed behind a contained API so that compile time selection was feasible.

2 Prior Work

There have been several other notable efforts to make schedulers more scalable and efficient for SMP and UP. These efforts were examined while ULE was under development and many of them had some influence in its design. ULE was intended to be a production quality modern scheduler from the start. As such evolution was preferred over revolution.

Several schedulers are discussed below. The 4BSD scheduler provided refinements over the traditional UNIX scheduler. It was the basis for FreeBSD's current scheduler which ULE used as a target to match for its interactive and nice behavior. UNIX System V release 4 solved the O(n) problem by using a table driven scheduler. This approach was later refined in Solaris[4] and other SVR4 derived operating systems. Linux has also recently received a totally rewritten event driven scheduler from which ULE's queuing system was derived.

2.1 4.3 BSD

The 4.3BSD scheduler is an adaptation of the traditional UNIX scheduler[1]. It provides excellent interactive response on general purpose time sharing systems. It does not, however, support multiple scheduling classes or SMP. This scheduler is discussed in greater detail in [2].

The scheduler derives priorities from a simple estimation of recent CPU usage and the nice setting of the process. The CPU usage estimation, *estcpu*, is calculated as the sum of the number of ticks that have occurred while the process has been running. This value is decayed once a second by a factor that is determined by the current load average of the system. The value is also decayed when processes sleep and wakeup. This decay improves the priority of a process and may allow it to run.

The decay function is the primary reason that the scheduler performs poorly with large numbers of processes. In order for the scheduler to operate on processes which may have been sleeping, it must iterate over every process in the system to decay their *estcpu*. Without this regular decay, processes would not trigger any events that would affect their priority and so they would be indefinitely starved.

Contrary to many newer designs, there are no fixed time slices other than a round robin interval. Every 100ms the scheduler switches between threads of equal priority. This ensures some fairness among these threads.

Nice values have a fixed impact on the priority of a process. Processes with a high negative nice value can prohibit other processes from running at all because the effect of *estcpu* is not great enough to elevate their priority above that of the un-nice process.

Since this scheduler was developed before multiprocessors were common, it has no special support for multiple CPUs. BSD implementations that still use this scheduler simply use a global run queue for all CPUs. This does not provide any CPU affinity for threads. The lack of CPU affinity can actually cause some workloads to complete slower on SMP than they would on UP!

2.2 FreeBSD

FreeBSD inherited the traditional BSD scheduler when it branched off from 4.3BSD. FreeBSD extended the scheduler's functionality, adding scheduling classes and basic SMP support.

Two new classes, real-time and idle, were added early on in FreeBSD. Idle priority threads are only run when there are no time sharing or real-time threads to run. Real-time threads are allowed to run until they block or until a higher priority real-time thread becomes available.

When the SMP project was introduced, an interrupt class was added as well. Interrupt class threads have the same properties as real-time except that their priorities are lower, where lower priorities are given preference in BSD. The classes are simply implemented as subdivisions of the available priority space. The time sharing class is the only subdivision which adjusts priorities based on CPU usage.

Much effort went into tuning the various parameters of the 4BSD scheduler to achieve good interactive performance under heavy load as was required by BSD's primary user base. It was very important that systems remain responsive while being used as a server.

In addition to this, the nice concept was further refined. To facilitate the use of programs that wish to only consume idle CPU slices, processes with a nice setting more than 20 higher than the least nice currently running process will not be permitted to run at all. This allows distributed programs such as SETI or the rc4 cracking project to run without impacting the normal workload of a machine.

2.3 System V Release 4

The SVR4 scheduler provided many improvements over the traditional UNIX scheduler[3]. It added several scheduling classes which include time sharing, real-time, idle and fixed priorities. Each class is supported by a run time selectable module. The entire scheduler was also made order one by a table driven priority system.

The individual classes map their own internal priorities to a global priority. A non-class-specific portion of the scheduler always picks the highest priority thread to run.

Fairness was implemented by penalizing threads for consuming their full time slice without sleeping and applying some priority boost for sleeping without using a full time slice. The current priority of the thread is used as an index into a table that has one entry for each event that will affect the priority of a thread. The entry in the table determines the boost or penalty that will be applied to the priority.

In addition to this the table provides the slice size that a thread will be granted. Higher priorities are granted smaller slices since they are typically reserved for interactive threads. Lower priority threads are likely to be CPU bound and so their efficiency is increased by granting them a larger slice.

2.4 Linux

In January 2002 Linux received a totally rewritten scheduler that was designed with many of the same goals as ULE[5]. It features O(1) algorithms, CPU affinity, per CPU run queues and locks and interactive performance that is said to be on par with their earlier scheduler.

This scheduler uses two priority array queues to achieve fairness. After a task exhausts its slice it is placed on an expired queue. There is an active queue for tasks with slices. The scheduler removes tasks from this queue in priority order. The scheduler switches

between the two queues as they are emptied. This prevents high priority threads from starving lower priority threads by forcing them onto the secondary queue after they have run. This mechanism is covered in more detail in the ULE section.

The priority is determined from a simple *estcpu* like counter that is incremented when a task sleeps and decremented when it runs. Slice sizes are dynamic with larger slices given to tasks with higher priorities(numerically lower). This is contrary to many scheduler designs which give smaller slices to higher priority tasks as they are likely to be interactive. Lower priority threads are given larger slices so that negative nice values will positively affect the CPU time given to a thread.

3 The ULE Scheduler

The ULE scheduler was designed to address the growing needs of FreeBSD on SMP/SMT platforms and under heavy workloads. It supports CPU affinity and has constant execution time regardless of the number of threads. In addition to these primary performance related goals, it also is careful to identify interactive tasks and give them the lowest latency response possible.

The core scheduling components include several queues, two CPU load-balancing algorithms, an interactivity scorer, a CPU usage estimator, a slice calculator, and a priority calculator. These components each are discussed in detail in the following sections.

3.1 Queue Mechanisms

Each CPU has a kseq (kse queue) structure, which is named after a component in FreeBSD's new threading architecture. Each kseq contains three arrays of run queues that are indexed by bucketed priorities. Two run queues are used to implement the interrupt, real-time, and time sharing scheduling classes. The last is for the idle class. In addition to these queues the kseq also keeps track of load statistics and the current nice window, which will be discussed in the section on nice calculation.

Since ULE is an event driven scheduler there is no periodic timer that adjusts thread priority to prevent starvation. Fairness is implemented by keeping two queues; current and next. Each thread that is granted a slice is assigned to either the current queue or the next queue. Threads are picked from the current queue in priority order until the current queue is empty. At this time the next and current queues are switched. This guarantees that each thread will be given use of its slice once every two queue switches regardless of priority.

A thread is assigned to a queue until it sleeps, or for the duration of a slice. The base priority, slice size, and interactivity score are recalculated each time a slice expires. The thread is assigned to the current queue if it is interactive or to the next queue otherwise. Inserting interactive tasks onto the current queue and giving them a higher priority results in a very low latency response.

Interrupt and real-time threads are also always inserted onto the current queue, as are threads which have had their priorities raised to interrupt or real-time levels via priority propagation. Without this, a non-interactive thread on the next queue that is holding some important resource, such as the giant lock, could prevent a high priority thread from running.

The idle class has its own separate queue. This queue is checked only when there are no other runnable tasks. Idle tasks are always inserted onto this queue unless they have had their priority raised via priority propagation.

Traditionally in BSD a running thread is not on any run queue. ULE partially preserves this behavior. The running thread is not linked into a run queue, but its load and nice settings are accounted for in the kseq. Accounting for the load is important for kseq load-balancing. It is desirable to distinguish between a kseq that is busy running one thread and one that has no load at all when picking the least loaded kseq in the system.

3.2 Interactivity Scoring

The interactivity scoring mechanism has a substantial affect on the responsiveness of the system. As a result of this, it has the most impact on user experience. In ULE the interactivity of a thread is determined using its voluntary sleep time and run time. Interactive threads typically have high sleep times as they wait for user input. These sleeps are followed by short bursts of CPU activity as they process the user's request.

Voluntary sleep time is used so that the scheduler may more accurately model the intended behavior of the application. If involuntary sleep time was taken into consideration, a non-interactive task could achieve an interactive score simply because it was not permitted to run for a long time due to system load.

The voluntary sleep time is recorded by counting the number of ticks that have passed between a sleep() and wakeup() or while sleeping on a condition variable. The run time is simply the number of ticks while the thread is running.

An interactivity score is computed from the relationship between the sleep time and run time. If the sleep time exceeds the run time, the interactivity score is the ratio of sleep to run time scaled to half the interactive score range. If the run time exceeds the sleep time, the ratio of run time to sleep time is scaled to half of the range and then added to half the range. This equation is illustrated in Figure 1 below.

$$m = \frac{(Maximum\ Interactive\ Score)}{2}$$

$$if(sleep > run)\ score = \frac{m}{(\frac{sleep}{run})}$$

$$else\ score = \frac{m}{(\frac{run}{sleep})} + m$$

Figure 1: Interactivity scoring algorithm

This produces scores in the lower half of the range for threads whose sleep time exceeds their run time. Scores in the upper half of the range are produced for threads whose run time exceeds their sleep time.

These two numbers are not allowed to grow unbounded. When the sum of the two reaches a configurable limit, they are both reduced to a fraction of their values. This preserves their relative sizes while remembering only a fraction of the thread's past behavior. This is important so that a thread which changes from interactive to non-interactive will quickly be discovered. This is actually quite a common case since many processes are forked from shells that have interactive scores.

Keeping too large or small of a history yields poor interactive performance. Many interactive applications exhibit bursty CPU usage. Some applications may perform expensive actions as a result of user input. For example, rendering an image or a web page. If the application's history of waiting for user input was not retained for long enough, it would immediately be marked as non-interactive when it did some expensive processing. If the time spent sleeping was remembered for too long, a previously interactive process would be allowed to abuse the system.

The scheduler uses the interactivity score to determine whether or not a thread should be assigned to the current queue when it becomes runnable. A threshold on the interactivity score is set and threads which score below this threshold are marked as interactive.

This threshold and the amount of history kept are two of the most important factors in keeping the system interactive under load. If the threshold is set too low, expensive interactive applications such as graphical editors, web browsers, and office suites would not be marked as interactive. If it is set too high, compilers, scientific applications, periodic tasks etc. would be marked interactive.

3.3 Priority Calculator

The priority is used in ULE to indicate the order in which threads on the run queue should be selected. It is not used to implement fairness as it is in some other schedulers. Only time sharing threads have calculated priorities; the rest are assigned statically.

A fixed part of the priority range in FreeBSD is used for time sharing threads. ULE uses the interactivity score to derive the priority. After this the nice value is added to the priority, which may actually lower the priority in the case of negative nice values.

This generally gives interactive tasks a chance to run sooner than non-interactive tasks when they are placed on the same queue. It may, however, allow for non-interactive negative nice processes to receive a better priority than an interactive, yet still expensive, process. This is desirable so that nice may have a positive effect on response time as well as the distribution of CPU time.

3.4 Nice Impact / Slice Calculator

One of the more difficult decisions in ULE was how to properly deal with nice. This was difficult because, as was previously discussed, ULE runs every thread at least once per two queue switches and, given certain conditions, some threads should not run at all. The final implementation of nice involves a moving window of nice values that are allowed slices.

ULE keeps track of the number of threads in the kseq with each nice value. It also keeps the current minimum nice value, that is, the least nice process. To be compatible with the 4BSD scheduler ULE only allows threads that have nice values within 20 of the least nice thread to obtain slices. The remaining threads receive a zero slice and are still inserted onto the run queue. When they are selected to be run their slice is reevaluated and then they are placed directly onto the next queue.

The threads that are within the nice window are given a slice value that is inversely proportional to the difference between their nice value and the least nice value currently recorded in the kseq. This gives nicer threads smaller slices which very granularly and

deterministically defines the amount of CPU time given to competing processes of varying nice values.

On x86, FreeBSD has a default HZ of 100. This leaves us with a minimum slice value of 10ms. ULE chooses 140ms as the maximum slice value. Larger values are impractical since they would unacceptably increase the latency for non-interactive tasks. This means that the least nice thread will receive 14 times the CPU of the most nice thread that is still granted a slice. Since there are 40 nice values, differences of less than three do not impact the slice size although they do have a minimal affect on priority.

Interactive tasks receive the minimum slice value. This allows us to more quickly discover that an interactive task is no longer interactive. The slice value is meaningless for non-time-sharing threads. Real-time and interrupt threads are allowed to run so long as they are not preempted by a higher priority real-time or interrupt thread. Idle threads are allowed to run as long as there are no other runnable threads in the system.

3.5 CPU Usage Estimator

The CPU usage estimator is used only for ps(1), top(1), and similar tools. It is intended to show roughly how much CPU a thread is currently using. Often summing all of the CPU usage percentages in a system can yield numbers over 100%. This is because the numbers are smoothed over a short period of time.

ULE keeps track of the number of statistics clock ticks that occurred within a sliding window of the thread's execution time. The window grows up to one second past a threshold and then is scaled back down again. This process keeps the ratio of run time to sleep time the same after scaling while making the actual preserved count smaller. This is so that new ticks or sleeps will have a greater impact than old behavior since old behavior accounts for a smaller percent of the total available history.

Keeping ULE O(1) required implementing a CPU usage estimator that operated on an event driven basis. Since a thread may go to sleep for a long time it will have no regular event to keep its CPU usage up to date. To account for this, a hook was added to the scheduler API that is used whenever the CPU usage is read. This hook adjusts the current tick window and tick counts to keep the statistics sane.

There is a rate limiter on this hook to prevent rounding errors from eroding away some of the CPU usage. Before this limit was implemented, top(1) caused the CPU usage to reach zero if it was constantly refreshed.

3.6 SMP

The primary goal of ULE on SMP is to prevent unnecessary CPU migration while making good use of available CPU resources. The notion of trying to schedule threads onto the last CPU that they were run on is commonly called CPU affinity. It is important to balance the cost of migrating a CPU with the cost of leaving a CPU idle.

Modern processors have various large caches that have significant impacts on the performance of threads and processes. CPU affinity is important because a thread may still have data in the caches of the CPU that it last ran on. When a thread migrates to a new CPU, not only does it have to load this data into the cache of the CPU that it is running on but cache lines on the previous processor must be invalidated.

ULE supports two mechanisms for CPU load-balancing. The first is a pull method, where an idle CPU steals a thread from a non-idle CPU. The second is a push method where a periodic task evaluates the current load situation and evens it out. These two mechanisms work in tandem to keep the CPUs evenly balanced with a variety of workloads.

The other situation that ULE takes into consideration is SMT (Symmetric Multi-Threading), or Hyper-Threading as it is called on Intel Pentium4 CPUs. ULE treats SMT as a specific case of NUMA (Non-Uniform Memory Architecture). Based on information provided by the machine dependent code, ULE is able to make scheduling decisions where migrating between different sets of CPUs have different costs and all CPUs are not equal.

3.6.1 Pull CPU Migration

Pull CPU migration is designed to prevent any CPU from idling. It is mostly useful in scenarios where you have light or sporadic load, or in situations where processes are starting and exiting very frequently.

With small numbers of processors it is less expensive to lock the run queue of another processor and check it for runnable threads than it is to idle. Because of this, ULE simply implements CPU migration by checking the queues of other CPUs for runnable threads when a CPU idles.

All of the available kseqs are compared and the highest priority thread is selected from the most loaded kseq. Some alternate algorithms were considered, such as selecting the thread that ran the least recently. Some attempt at balancing the interactive and non-interactive

load was also made. Neither of these two approaches showed any measurable gain in any test workloads.

This pull method ends up being effective for short running, high turnover tasks such as batch compiles. It is not quite as good at evenly distributing load for very short lived threads as having a single run queue. The other advantages of the scheduler should outweigh this minor disadvantage.

3.6.2 Push CPU Migration

The pull model for CPU migration is not effective if all CPUs have some work to do but they have an uneven distribution of work. Without push migration, a system with several long running compute-bound tasks could end up with a severe imbalance. For example, it takes just one thread using 100% of the CPU to prevent the pull method from working even if the other processors have many runnable threads.

To prevent this scenario ULE has a timeout that runs twice a second. Each time it runs it picks the two most unbalanced kseqs and migrates some threads to the kseq with the lowest load. If the kseqs are imbalanced by only one thread one thread is still moved from one kseq to the next. This is useful to ensure total fairness among processes.

Consider a two processor system with three compute-bound processes. One thread has an affinity for the first processor while the remaining two threads have an affinity for the second. If we did not periodically move one thread, the thread on the first processor would complete twice as quickly as the threads on the second!

This timeout does not attempt to completely balance all CPUs in the system. There are two advantages to this. The first is that FreeBSD is likely to see more dual processor systems than any other SMP configuration. So the algorithm is simple and perfectly effective for the common case.

On systems with more than two processors it will only take slightly longer to balance the CPUs. This is advantageous because CPU load is very rarely regular. Attempts at over aggressive balancing are likely to ruin caches and not resolve real load imbalances.

3.6.3 SMT Support

Symmetric Multi-Threading presents the scheduler with a slight variation on SMP. Since the logical CPUs in a SMT system share some resources they are not as powerful as another physical CPU. To take full advantage of SMT, the scheduler must be aware of this.

ULE effectively takes advantage of the lack of penalty for migrating threads to a CPU on the same core as well as treating the logical CPUs as less capable than true physical cores. ULE accomplishes this by mapping multiple logical CPUs to the same kseq. This ensures that in a system with multiple SMT capable cores the load-balancing algorithms will prefer to distribute load evenly across groups of CPUs. Since logical CPUs on the same core typically share cache and TLB resources a thread may have run on either without paying any penalty for migration.

A more naive implementation would treat all of the cores as equal. This could lead to many logical cores on the same CPU being used while other physical processors with more resources go unused.

SMT introduces a concept of non-uniform processors into ULE which could be extended to support NUMA. The concept of expressing the penalty for migration through the use of separate queues could be further developed to include a local and global load-balancing policy. At the time of this writing, however, FreeBSD does not support any true NUMA capable machines and so this is left until such time that it does.

4 Late: A Workload Simulation Tool

Late was developed as a means for creating synthetic CPU intensive workloads for use in analyzing scheduler behavior. It collects a variety of metrics that are useful when comparing scheduler implementations.

Late works by running and sleeping over a configurable period. CPU intensive work is simulated by using memcpy to transfer data between two fixed buffers. There is a calibration loop that is run on an idle system prior to any test that determines the number of loops required to occupy the processor for a number of microseconds. The sleep is simply implemented via nanosleep(3).

The time required to execute each work period is averaged out over the course of a run. The maximum time is also kept. The difference between the requested wakeup time from nanosleep(3) and the actual wakeup time is also averaged. These statistics may optionally be reported once a second along with the current priority of the late process. At the end of each run, CPU time, real time, sleep time, and %CPU are displayed. The number of voluntary and involuntary context switches are also displayed when they are provided by the operating system.

Late has the ability to wait for a SIGUSR1 before performing its configured task. This makes setting up

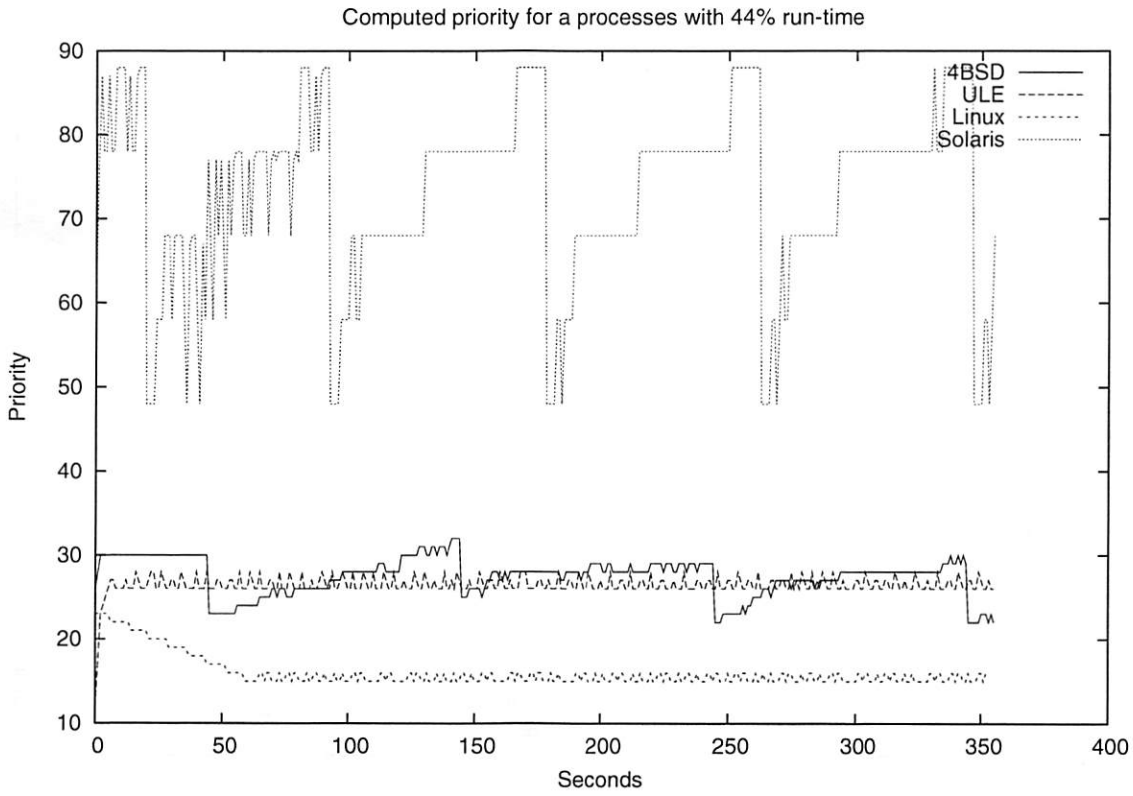

Figure 2: This graph depicts the fluctuations in priority for a process with a constant run-time. The priorities assigned relative to the other schedulers are meaningless. The function of priority and mechanisms of its assignment, however, become evident when it is shown over time.

and coordinating several late processes easy to do from a shell script. Once the late process is permitted to run it may also set its nice value for nice related tests. Setting the nice value from within the late process is a good way to guarantee that all processes were given a chance to proceed up to that point, which they may not have if their priority was too low from the start.

Using these simple facilities late processes can be combined to illustrate various scheduler behaviors by simulating different workloads. While this tool proved to be indispensable during development and for producing benchmarks, it is no substitute for real user experience.

5 Benchmarks

The benchmarks were all created using late to generate synthetic workloads. Combinations of processes simulating compute-bound tasks, batch compiles, vi editors and web browsing were used. A mix of sleep and run time for each simulated application was determined through observing typical runs of these processes on the test system. The benchmarks are intended to give an indication of how a scheduler might perform under load. Due to their artificial nature they are not an absolute representation of real user experience.

The system running the tests was a dual Athlon MP 1900+ with 512mb of memory. Three operating systems were installed; FreeBSD 5.1-CURRENT with the 4BSD and ULE schedulers, Mandrake Linux 9.1 with the 2.5.73 kernel, and Solaris/x86 9.1. The second processor was disabled during each test.

To measure interactivity, all tests, except where noted, ran 4 simulated vi sessions and 2 simulated Mozilla sessions. This mix was chosen to emulate typical use of a server or workstation machine with an administrator or user present in a graphical environment. This should be fairly representative because shells and vi sessions exhibit similar CPU usage characteristics and a graphical administration tool, word processor, or image editor would be represented by the Mozilla simulation. It is important to have expensive interactive applications as well as cheap ones to mimic the use of systems as workstations.

Late keeps track of the time it took for the process to wake up after a timer fired and the time taken to

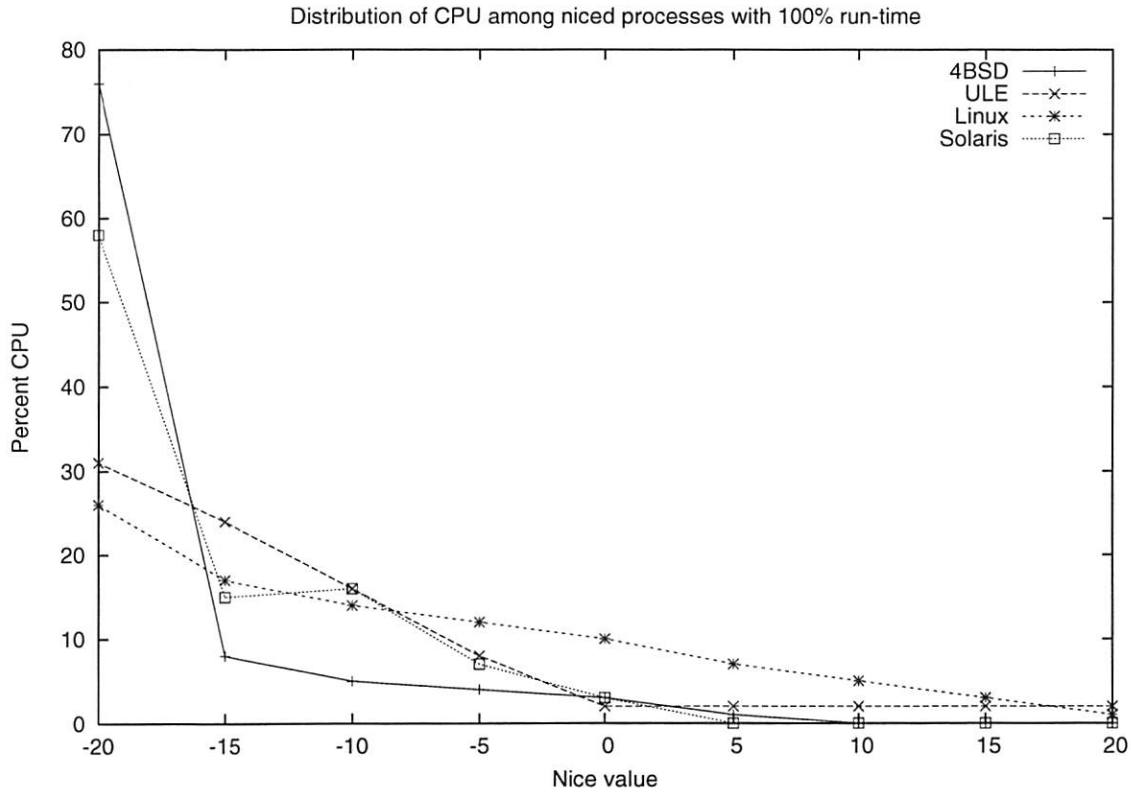

Figure 3: This graph depicts the distribution of CPU granted vs. nice assignment. Two important things are visible here. Firstly, the Solaris and 4BSD schedulers, which always run the highest priority task, give an non-proportional amount of CPU time to nice -20 processes. Secondly, Linux has no cut-off after which processes receive no CPU time.

execute the workload. These two are summed together to indicate the response time for user input in the various interactive tasks.

For the interactivity graphs, the sum of the latency for all simulated interactive processes is graphed. This is not significantly different from the graph of any individual process in most cases. It does, however, widen the gap between the different schedulers in some tests which leads to more easily interpreted results.

5.1 Priority Calculations

In this test a single late process was used. This process runs for 40ms and then sleeps for 50, and thus simulates 44% CPU utilization. Various methods were used to collect the priority of this late process once a second over a 6 minute run. It is important to note that the relative priorities between schedulers are meaningless. We are only concerned with the priority of the process relative to the others in the system and how it is adjusted with CPU usage. This is useful to understand when analyzing the behavior of subsequent tests. The results of this test are shown in figure 2 above.

The periodic decay loop is evident in the cyclic priority assignments of the 4BSD scheduler. This is ensures that even high priority processes will eventually get CPU time by decaying them to a lower priority. Solaris mimics this behavior even without a periodic decay algorithm. It is likely that once the priority reaches a certain value it is reduced significantly to ensure that the process gets some time to execute and thus mimicking the behavior of the 4BSD scheduler.

ULE and Linux do not base their fairness on priority and so they tend to find a stable state. ULE starts at an interactive priority as it inherited it from the shell and then moves up to the priority determined by the CPU utilization of the process. The priority in ULE oscillates around a small range as the process runs and sleeps but it averages out to a level line.

Linux steadily decays the priority until it hits the minimum value. This means that this process, which is using 44% of the CPU, reaches the best user priority possible without a negative nice setting, after approximately 60 seconds. Due to the simple way that Linux tracks CPU utilization, given enough time the

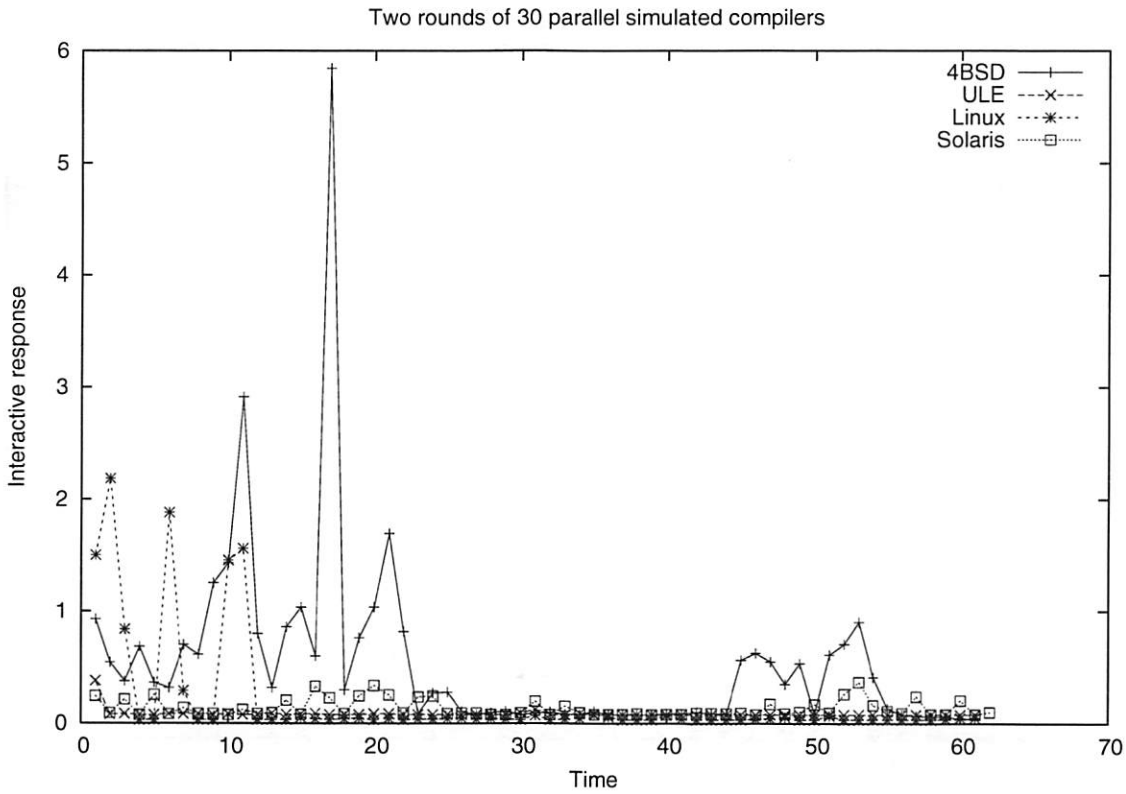

Figure 4: This graph shows the sum of the response time for several interactive applications while the system is running 2 batches of 30 parallel compilers. All of the schedulers do fairly well once the test has been running for some time although they vary significantly in how long they take to stabilize.

priority of a process with a very regular run-time will reach the minimum priority if it runs less than it sleeps and the maximum priority otherwise. This leads to some pathological scenarios which are demonstrated later.

5.2 Effects of Nice

The test in Figure 3 shows the distribution of CPU time among several nice processes running simultaneously. Nice values from 20 to -20 in steps of 5 were chosen to illustrate a wide range of activity on the machine. The results of these tests only illustrate differences in behavior. There is no correct or well defined way to handle nice values.

The processes in this test were never yielding the CPU. This illustrates an important feature of all the schedulers other than Linux. Processes that exceed some nice threshold are not allowed to run at all when there is other load on the system. This was most useful before idle classes were introduced although it is still a common practice to nice a process so that it will take only idle cycles. Curiously, Linux does not have an idle scheduling class either and so this class of applications will always consume CPU time on a Linux machine regardless of load.

Here both of the older schedulers show a clearer bias for -20 processes than the newer two. This is probably more of an artifact of how difficult it is for processes to travel 20 priority points from their CPU usage than it is any design goal. Deriving nice based CPU distribution from the slice size yields a much more even slope in both ULE and Linux.

This data looks somewhat irregular for ULE. The processes that were past the cutoff point were still given slightly less than 1% of the CPU. This is due to an interaction between the simulated workload and ULE's queuing mechanism. The nice late processes were allowed to run for one slice after the un-nice processes exited.

5.3 Interactivity Tests

Figures 4 and 5 illustrate the sum of the response time for several interactive applications. These tests are designed to illustrate how responsive the system remains under several workloads. The synthetic benchmarks do a reasonable job of pointing out what

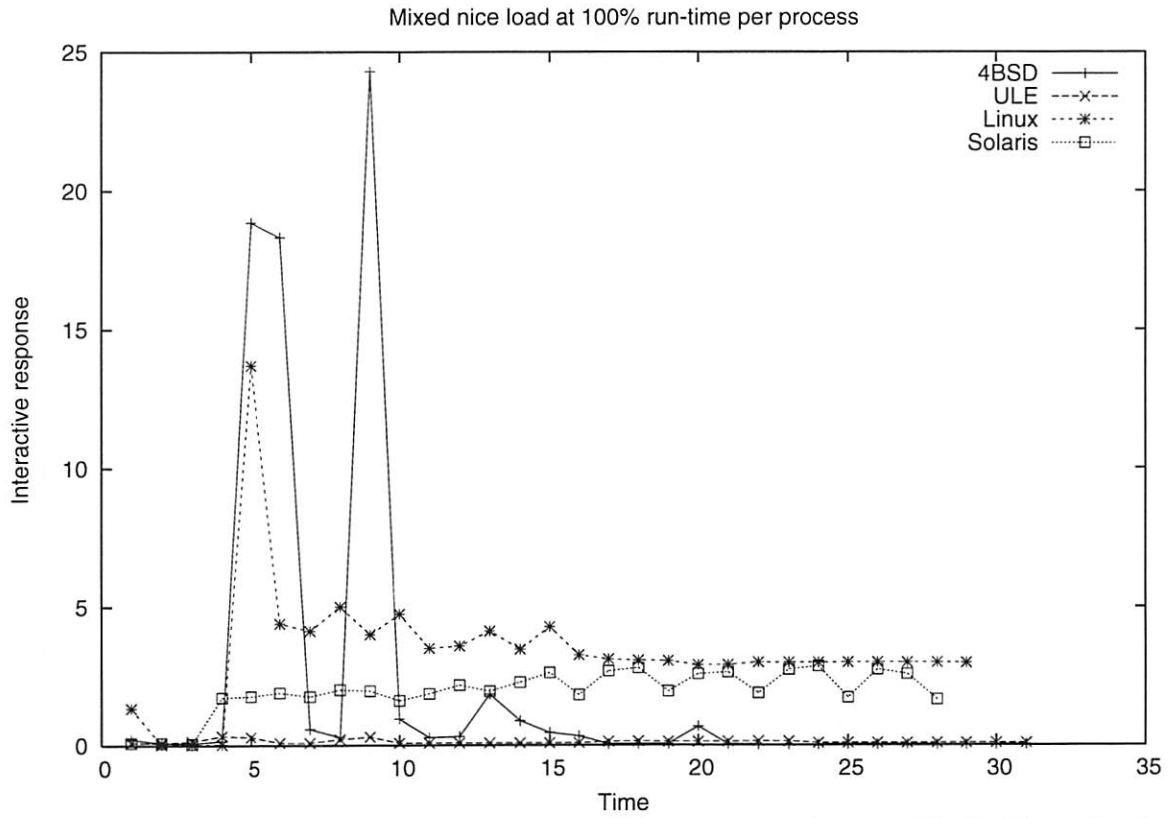

Figure 5: The nice test from Figure 3 was run while measuring interactive latency. All schedulers other than ULE allow a few negative nice processes to have an impact on interactive response.

areas may be problematic. Subjective tests have shown, however, that minor problems exposed by the synthetic tests typically result in much worse interactive behavior than is suggested.

The data for figure 4 was gathered during a test that simulated two sets of 30 parallel compiles. The second set was gated by the completion of all tests in the first set as you would see with a typical build dependency. This illustrates a more realistic workload than the simulations involving nice processes. We can see that all of the schedulers do fairly well once the compilers have been running for a short while. This is due to the time it takes the schedulers to learn the behavior of the compiler processes.

The 4BSD scheduler has the worst spikes due to the high priority inherited from the shell and the time it takes to overcome that. ULE is hardly visible on this graph because it has no spikes in interactive response time and very regularly exhibits almost no latency whatsoever. Solaris has only a few small bursts of latency throughout the test. Linux has a few significant peaks in the beginning and then settles out and performs quite well. Only 4BSD and Solaris show signs of the second batch of compilers starting near 45 seconds. Linux and ULE both distribute more of the learned behavior from child processes to parents when they exit.

Figure 5 was generated from data gathered during the nice related test illustrated above. The nice processes began running 3 seconds after the interactive tasks. In this graph ULE is as responsive as it is under no load. 4BSD is generally responsive after the test is underway but it suffers from several extreme spikes in latency at the begining.

Solaris is the next most responsive after 4BSD. Although an average response time of 2.5 seconds is not acceptable for editor use, it would be enough to use a shell to kill the offending processes. Linux has the worst average latency in this test. This is probably due to the effect of nice on the priority of the process and the time slice granted. The negative nice processes regularly exceed the priority of the interactive processes and starve them for CPU. Even cheap, extremely interactive processes, such as vi and shells, suffer as a result.

All of the schedulers other than ULE exhibit such extreme latency in responding to the interactive tasks

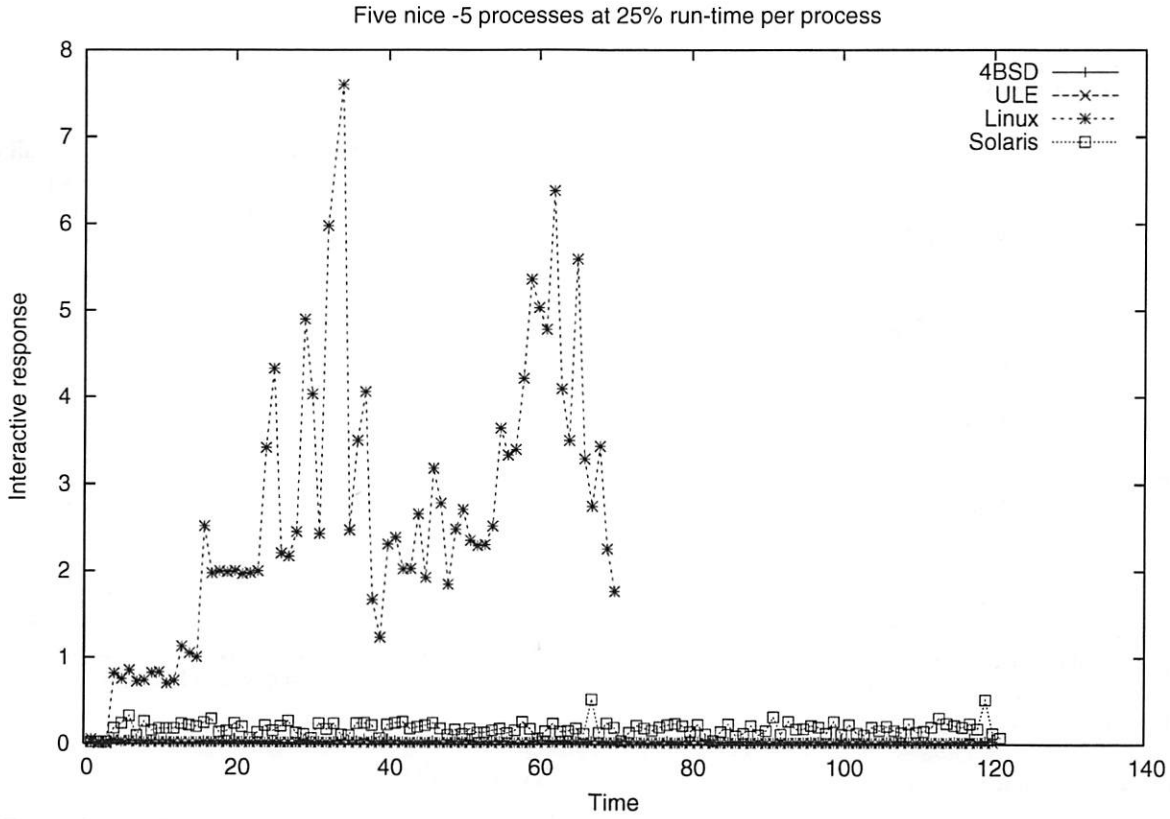

Figure 6: A pathological case in the Linux scheduler is exposed. Priorities for nice -5 processes using 25% of the CPU severely exceed the priority of a simulated vi session.

that they were not able to complete their runs within the time constraints provided. This is why Linux and Solaris seem to have lost several seconds from their graph. ULE stands out as having clearly lower response time and higher regularity than any other scheduler in this test.

Figure 6 illustrates a pathological case for the Linux scheduler which early versions of ULE fell victim to. The setup is 5 nice -5 processes each attempting to use 25% of the CPU. This over-commits the CPU by 25%, which should not be a problem. However, since Linux gradually reduces the priority until it hits the minimum, the nice value is enough to prevent even normal interactive tasks from running with reasonable latency. This was solved in ULE by using the interactivity scoring algorithm presented above.

5.4 Performance

Development of ULE thus far has primarily focused on the interactivity and 'nice' behavior of the scheduler. Now that these algorithms are stabilizing, the focus is shifting to performance tuning. An in-depth analysis of ULE's performance is not within the scope of this paper. However, some initial numbers can be obtained from running the tests performed above with SMP enabled kernels.

The parallel compile test completed four times faster on ULE than it did on 4BSD. This is entirely due to the effects of CPU affinity. This data will not be representative of the results seen with real compilers for several reasons. The synthetic load is entirely memory bound and heavily impacted by the CPU cache. Also, the amount of memory transferred is small and so it is likely that many, if not all, of the late processes can be held in the cache of the two CPUs simultaneously. Late also rarely enters the kernel to do more than a nanosleep(2). Real compilers often contend on kernel resources which reduce the possible parallelization and cause more frequent context switches.

While late demonstrates what is probably close to the best case for CPU affinity gains, early tests with apache may give more realistic results. ULE bests 4BSD by 30% on apache throughput benchmarks on a dual processor Xeon system. This is likely to be more representative of the gains that can be expected from CPU affinity in real applications.

6 Conclusions

While the design of ULE primarily had performance-related goals, it quickly became clear that the architecture provided advantages in other areas as well. Other attempts in the past, such as SVR4/Solaris, have sacrificed interactive behavior for order one execution time. ULE achieves both by borrowing a novel approach from the 2.5 Linux scheduler and the development of new algorithms surrounding that mechanism.

The advantages in interactivity come from several key differences over earlier schedulers. Firstly, interactivity, priority, and slice size are separate concepts in ULE. Other schedulers closely tie these three parameters and are left with many side effects. In ULE each one can be adjusted mostly independently from the rest so that desirable behaviors may be achieved. As a result of this, another one of the key advantages of ULE is possible. Nice is viewed only as a hint by the administrator for non-interactive jobs. As such, the system remains completely responsive despite the nice load. Livelock under nice load has been a constant problem for UNIX schedulers which ULE now avoids entirely.

While the benchmark results are encouraging, a significant amount of time must be spent with ULE in real environments before it will be accepted as the default scheduler by the FreeBSD community.

7 Future Work

ULE was written over a weekend and refined over the course of a year. The 4BSD scheduler, by comparison, has seen a decade of refinement in the FreeBSD project alone. Hopefully, much of that refinement was captured in the new design. Despite the differences in algorithms, ULE should represent another step in the evolution of the UNIX scheduler. Aside from further tuning for a wider variety of workloads, ULE has some areas that are still in need of more analysis and development.

The SMP load-balancing algorithms need to be studied in at least as much detail as the interactivity algorithms have been. They are currently quite primitive, although they may stay that way. Attempts at making them more intelligent have only led to minor improvements in some areas to the extreme detriment of others. This topic is worthy of a paper on its own.

Non-uniform CPU architectures are starting to become popular. Effective support for this in ULE will provide many users with a more compelling reason to switch. At this time enabling Hyper-Threading logical cores leads to worse performance than having them disabled. Recent changes to ULE close that gap but more work is required.

Finally, late has been a very useful tool for scheduler development. However, there are several behaviors of real applications that it fails to capture. As a result of this, its effectiveness in comparing schedulers is reduced. Adding support for variable and bursty workloads as would be seen by real users seems to be the next logical step. After this, giving it multi-threading capabilities would allow us to analyze methods of scheduling multiple threads within the same process.

Acknowledgments

Thank you Dave Anderson, Jagdeep Johal, Mike Karels, Jonathan Mini, Donn Seeley, and Donya Shirzad for providing feedback on this paper. Thanks to Matthew Dillon for his review of the early code and Steve Kargl and FreeBSD-current for their constant testing. Also, thank you Donya for putting up with my long nights writing this paper and hacking on FreeBSD.

Availability

ULE is available via the FreeBSD source repository in any branch after 5.0. Please refer to http://www.FreeBSD.org to obtain the source or an installable FreeBSD image.

Late is available via the FreeBSD source repository in any branch after 5.1. The test scripts are available along with late.

References

[1] M. J. Bach, "*The Design of the UNIX Operating System*", Bell Telephone Laboratories, Inc. (1986)

[2] M. McKusick, K. Bostic, M. Karels, & J. Quarterman, "*The Design and Implementation of the 4.4BSD Operating System*", Addison Wesley Publishing Company (1996)

[3] U. Vahalia, "*UNIX Internals The New frontiers*", Prentice Hall (1996)

[4] J. Mauro, R. McDougall, "*Solaris Internals Core Kernel Architecture*", Sun Microsystems, Inc. (2001)

[5] I. Molnar, Linux O(1) Scheduler design document, http://lxr.linux.no/source/Documentation/sched-design.txt?v=2.5.56

An Automated Binary Security Update System for FreeBSD

Colin Percival
Computing Lab, Oxford University
colin.percival@comlab.ox.ac.uk

Abstract

With the present trend towards increased reliance upon computer systems, the provision and prompt application of security patches is becoming vital. Developers of all operating systems must generally be applauded for their success in this area; systems administrators, however, are often found lacking.

Anecdotal evidence suggests that for FreeBSD much of the difficulty arises out of the need to recompile from the source code after applying security patches. Many people, after spending years using closed-source point-and-click operating systems, find the concept of recompiling software to be entirely foreign, and even veteran users of open source software are often less than prompt about applying updates. Providing these people with a binary option should significantly improve the rate at which security updates are applied.

This paper describes an automated system for building and distributing binary security updates for FreeBSD, and describes the challenges encountered. I also describe some of the limitations of this system, and discuss some possibilities for future work.

1 Introduction

Over the past few years, there has been a trend towards a much more rapid exploitation of security holes. It has been shown with honeypots that insecure systems are often compromised within days or hours of being connected to the internet [An02, Ho02]; it has even been suggested that a significant proportion of systems connected to the internet could be compromised by a sophisticated worm within 30 seconds [SPW02].

At the same time, there is a constant influx of new users into the FreeBSD community; even with detailed instructions, traffic on the FreeBSD mailing lists indicates that a large number of people find the task of applying security patches and rebuilding affected programs to be difficult and/or confusing. Given that releases are on average several months – and several security holes – old by the time they are installed, the possibility arises that a new user will find his system compromised before he has a chance to bring it up to date.

Furthermore, there are some circumstances where building from source is undesirable. Some embedded systems might lack sufficient disk space to store the entire source and object trees; some system administrators remove part or all of the build toolchain in an (arguably misguided) attempt to thwart any attempt to build a rootkit; and the purveyors of application-specific 'toasters' might very likely wish to keep the complexity of building from source entirely hidden from their users.

For these reasons, we believe that the provision of a system of binary security updates is absolutely critical.

2 Previous work

A large number of binary update systems have been created for various applications and operating systems, for both security updates and more general software updates. We first consider systems specific to security updates.

Between June 2001 and May 2002, many FreeBSD security advisories were accompanied by 'experimental binary upgrade' packages [SA02]. These were built by hand based on (human) consideration of which binaries should have been modified by a given source patch, and distributed in the standard FreeBSD package format. When a large number of binaries were affected (for example, if a library was modified) binary upgrade packages were not provided.

A similarly experimental, but rather more limited, system has been created for OpenBSD [Ga03]. Here, por-

tions of the 'world' are rebuilt according to instructions accompanying the official source patches, and a (hand-picked) subset of the files built are packaged into a compressed 'tar' archive, which is installed simply by extracting the new files over the old. Again, it does not appear that any attempt was made to handle patches affecting large numbers of binaries spread across the 'world'.

A more sophisticated approach was taken by a commercial service which currently provides binary updates for NetBSD [PST03]. Based on the MD5 [Ri92] digests of binaries pre- and post-patching, a list of potential distribuends is compiled. This list is then inspected by hand to remove files which are "modified but not related"; we will describe later how some binaries end up being modified even without any changes in the source tree. This hand-pruned set of binaries is then packaged into a shell script which provides the options of installing the new binaries, reverting to the previous binaries, et cetera.

Because these systems are specific to security updates, they all attempt to minimize the number of files updated, and they all include human participation in this effort. This raises a significant danger of error; even under the best of conditions, humans make mistakes, and the task of determining which files out of a given list had been affected by a given source code patch requires detailed knowledge of how the files are built. A good example of this is the SunRPC XDR library bug from March 2003 [CE03] – few, if any, people would have expected to find vulnerable XDR code in `/bin/mv` or `/bin/rm`. We argue therefore that building updates automatically has an advantage of correctness as well as an advantage of economy.

On the side of general binary updates, the field is more varied. Perhaps the best known of these is Microsoft's Windows Update, which distributes security updates, service packs, driver updates, and new versions of Microsoft 'middleware'; these are installed in the same manner as application software. Some application packaging tools also provide binary patch mechanisms; for example, InstallShield has an update service [IS03] which, depending upon the tool purchased, can replace an application entirely, distribute only modified files, or distribute only patches to the modified files.

The RedHat and Debian distributions of Linux both have binary update systems, up2date and apt-get respectively. Similar to these are portupgrade (which, as the name indicates, only upgrades software from the ports tree), and the FreeBSD binup project [Bi02], which aims to provide a general mechanism for all binary updates, but has unfortunately stalled due to a lack of developer time. All these tools work on the same general principle – everything is 'packagized', and the updating process consists simply of removing the old package and installing a new package.

3 Automated update building

In order to build binary updates without human intervention, we start with a very simple approach: Build the 'RELEASE' world in one directory, build the world based on the latest security patches in another directory, and compare. Any files which need to be included in the published update will have changed. Unfortunately, as noted earlier, the converse is not true: Some files will change every time they are built, even if they are built from the same source files; in FreeBSD 4.7, there are 160 such files, of which 128 are library archives.

Carefully examining the regions where these files differ shows the cause: They contain human-readable time and date stamps (hereafter we refer to these, along with user and host stamps, as 'build stamps'). Some of these are well known and serve obvious purposes: The kernel and boot loader, for example, display at startup the user, hostname, date, and time when they were built, and the library archives need to record timestamps so that their constituent object files can be accurately recreated; but other executables, such as those associated with perl, NTP, PPP, and ISDN, have build stamps without any apparent purpose.

In order to eliminate false positives introduced by these build stamps, we change our process as follows: We start by building the 'RELEASE' world twice, adjusting the clock to ensure that the date is different (some files contain the date, but not the time they were built), and then compare these two worlds in order to locate the build stamps. For binary files, we consider a build stamp to consist of a byte which differs between the two versions of the file and up to 128 'string' characters in either direction; for text files, we consider a build stamp to be a complete line which differs between the two versions of the file. Once we have located the build stamps, we build the new world and compare it to the release, excluding the regions previously marked as build stamps; any variation outside of those regions indicates that the relevant file needs to be distributed as part of a binary update. Finally, we rebuild the new world again and locate the new build stamps (this final step is necessary because any change to a binary is likely to move the build stamps.)

4 A few more complications

While the above procedure works for almost all files, a few need special treatment – usually in the form of cosmetic patches to the source tree. For some reason, fortune data files are randomized during the build process, even though fortune(6) already selects a fortune randomly. This causes the fortune data files to build differently every time; removing the randomization from the build process eliminates the variability without any noticeable effect.

On a related note, the compiler used for FreeBSD 4.x (gcc 2.95), in the rare case where it cannot find a programmer written global name in a given file, introduces a random string for this purpose. (In the FreeBSD 4.7 world, this only occurs when compiling the libobjc library.) Changing this behaviour to instead generate a global name by hashing the current path and the input filename removes the variability without affecting other functionality. [Si03]

When security patches are made to FreeBSD, it is standard practice to update a version string contained in the kernel. This has the advantage of making it apparent that the changes have been made; but it has the side-effect of causing the kernel to change when userland-only security fixes are applied. We override these changes.

Some of the documentation for groff uses the current date in examples; this would be handled properly as a timestamp, except that "March" is shorter than "February", and causes cascading differences in the line breaks. Modifying the examples avoids this problem.

Finally, some files are not entirely replaced during the build process: The directory used by info(1) and perl's perllocal.pod both have entries appended to them during the build process, but are never cleaned; the kernel building code keeps a count of how many times the kernel has been compiled; and some files (the kernel, modules, boot loader, and init) are backed up. Removing the info directory, perllocal.pod, the kernel compile counter, and the backup files eliminates the spurious variability which they introduce.

One additional complication is introduced by cryptographic export laws. Some files exist in multiple versions: Non-cryptographic, cryptographic, kerberos 4, and kerberos 5. We handle this by building the afflicted files, in all applicable forms, in separate directories.

Out of the patches necessary to work around these complications, only the one relating to fortune files has been incorporated into the main FreeBSD tree. Gcc and groff are 'contributed' code, and consequently local modifications are discouraged (we note, however, that the issue with gcc is likely to be corrected in a future version); and the question of kernel labelling resulted in a very lengthy debate when the current practice was first adopted and it seems unlikely to change now.

5 Distribution

Based on our generated list of which files need to be distributed, we generate an update index containing lines of the form

```
/path/to/file$oldhash$newhash
```

where `/path/to/file` is the full path to the file being updated, `oldhash` is the MD5 hash [Ri92] of the old version being replaced, and `newhash` is the MD5 hash of the new version being installed. Note that updating one file could result in several associated lines, one for each 'old version'; to handle this, we keep a list of all 'valid' old hashes by starting with the hash values from the published binary release and adding the hashes of any new files we distribute.

Along with the update index is distributed a 2048 bit public RSA key, the MD5 hash of the update index signed with the private part of the RSA key, the new versions of the files, identified by their MD5 hashes, and binary diffs generated with BSDiff [Pe03], identified by the MD5 hashes of the old and new versions. Note that these files, once created, are entirely static.

Given that the update index contains file hashes and the update index is signed, the only step which needs to be performed securely is the publication of the public key. This is done by verifying the MD5 hash of the public key; at present, a configuration file is distributed with the client software which includes the hash of a key belonging to the author (the mechanics of securely distributing application software is a bootstrapping issue and outside the scope of this paper); anyone else using this code to publish their own binary updates would naturally have to distribute their own key.

Everything else can be done insecurely: The update files can be distributed over insecure HTTP, can be mirrored easily, and can be transported via sneakernet to update a system which has no internet connection at all. This

also has the advantage of allowing updates to be built on a system which is physically disconnected from the outside world, with source patches carried in and published updates carried out manually.

6 Installation

Machines attempting to update themselves first download the RSA public key and verify that it has the correct MD5 hash; the update index is then downloaded, and the signature is verified. For each line in the update index, the MD5 hash of the currently installed file is then computed; if it matches the `oldhash` value contained in the update index, the binary diff is downloaded and the new version of the file is generated; as a backup method, if the file generated from the binary diff does not have the correct hash, the entire new file is downloaded (and verified to have the correct hash).

This use of binary diffs provides a remarkable reduction in bandwidth usage. Updating a typical installation of FreeBSD 4.7 (specifically, one which includes cryptography, but does not include either version of Kerberos) to include all the applicable security fixes as of mid-June 2003 involves replacing 97 files which total 36MB. The binary diffs for these total 621kB, a reduction by a factor of 58. Even if all the HTTP/TCP/IP overhead is added, the total bandwidth required for updating such a system is under 1.6MB – less than half of the 3.6MB used by cvsup [Po02] when performing the same update on the source tree. Indeed, based on an estimate from Netcraft that there are between 50 and 60 thousand publicly accessible web servers running FreeBSD worldwide [Pr03], and data from FreshPorts [La03] which suggests that web servers constitute slightly less than half of the machines running FreeBSD around the world, we believe that it would be possible for a single low-end server to provide binary updates to every FreeBSD system in the world within a single day.

After the updated files have been fetched and/or generated via patches, the updates are installed by backing up the old files and moving the new files over the old (subject to maintaining permissions, ownership, and file flags). Any supplemental tasks necessary for the updates to take effect (restarting daemons, recompiling staticly linked application software which uses modified libraries, and/or rebooting) is left to the system administrator.

It is important to note that this process is entirely stateless; no database is kept of which updates have been installed – instead, at each point, the currently installed files are examined to determine if they are 'old'. There is no mechanism for installing some, but not all, available updates – we assume that nobody would wish to patch some, but not all, security holes; indeed, there is no concept of an individual 'update'. Since the updates are produced by comparing the results of builds at various points along a security branch, there is no mechanism for identifying which particular security advisory corresponds to particular binary changes (unless, of course, there has only been one advisory in the applicable time window). Any administrator wishing to verify that he has not forgotten to update a system must simply run the client software; indeed, we encourage all potential users (i.e. people who started from a binary install and have not recompiled any part of the world) to set a cron job to run the update client.

7 Caveats

There are a few problems with the approach we take. First, because we rely upon the MD5 hash of currently installed files to determine which files need to be updated (and, in the case of export differences, which new version should be installed), any variation in the installed files will result in updates not being performed. This means that the set of potential users is restricted to those who have performed a binary install and not recompiled any FreeBSD files. We do not consider this to be a serious limitation, considering that our stated target audience is those users who are unable or unwilling to recompile from source.

Another limitation arises from the fact that we only *replace* files, rather than adding or removing them. This immediately means that this system is restricted to updates within a single version – while the effect of a security patch will be to modify some files, upgrading to a new version would require installing entirely new files. Upgrading from one FreeBSD release to another can already be performed simply by performing a binary install from the published release images.

A more serious issue arises with the kernel: We can only provide updates for the GENERIC kernel. While this may be sufficient for some users, a very obvious class exists for whom this is not sufficient – those with multiprocessor systems. We suggest therefore that it would be a Good Thing if FreeBSD releases also included at least a GENERIC-MP kernel, identical to the GENERIC

kernel except for the addition of multi-processor support; indeed, it might be adviseable to add 'bloat' to the GENERIC kernel in the interest of reducing the probability that someone would be required to build a custom kernel – noting, of course, that kernel modules can be updated, so features which can be fully supported via modules would not need to be compiled into the kernel.

Perhaps the most serious issue arises with configuration files and metadata. There are circumstances where a security update might need to change an option in a (user-serviceable) configuration file, or change the ownership, permissions, or flags on a file. This could be handled by transmitting patches for text files rather than simply distributing the new version, and recording which patches have already been applied; changes in ownership, permissions, or flags could be handled in a similar manner. However, such a mechanism would carry with it a considerable risk of damaging a customized configuration; consequently we feel that the principle of least astonishment requires that such (rare) fixes be left up to a human administrator.

A final issue arises from the use of the MD5 message digest. Although no collisions have been found, there is an ongoing attack [NP03]; indeed, it has been recommended for many years that MD5 not be used for applications where collision resistance is required [Ro96]. We note that even given the ability to compute MD5 collisions an attack would be very difficult, since it would require that specially crafted source code be introduced into the security branch; consequently, we consider the risk introduced by using MD5 to be negligible, and justifiable in light of the lack of stronger hash programs in the base FreeBSD distribution.

8 Future work

It seems very likely that the same approach, and much or all of the same code, can be used for building binary security updates to other BSD operating systems. There would, almost certainly, be a different set of patches necessary to remove spurious variabilities; but it would be very surprising if those necessary patches could not easily be found.

On the other hand, for various reasons it seems unlikely that this same approach could be applied to software from the FreeBSD ports tree. First, while pre-built packages are available, most people build ports from source; as noted earlier, this makes it impossible to provide updates. Second, there is no "security branch" for the ports tree; consequently, updating from one version to another is quite likely to involve adding or removing files, which is beyond the scope of this system. Third, the large number of ports, the time necessary for building them, and the (quite common) cases where some ports cannot be successfully built would all contribute to making such an attempt a logistical nightmare. We note, in any case, that portupgrade already makes the updating of ports quite simple.

On the other hand, this tool is ideally suited to self-contained "toasters". Providing that a vendor can construct an automated mechanism for building all installed software (operating system and software), binary updates can be built and distributed in exactly the same manner as for an operating system alone.

One possible future modification would be to keep a 'clean' copy of configuration files, and use mergemaster(8) to merge any changes if the configuration files were changed (in the same manner as when recompiling the entire world). This would not be automated – merging any changes would require the intervention of the system administrator – but it would at least be a step in the right direction.

The most interesting possibility, however, is for having several machines build updates and cross-sign. This would be far from trivial, since each machine would (due to the build stamps) produce different updates (the same files would be modified, but the new values would be different). However, it should be possible for each machine to fetch the updates built by each of the others and remove the build stamps before comparing; in this manner, they could each verify that each others' builds were identical up to (security-irrelevant) build stamps. Client machines could then be configured with a list of trusted keys, and could require that a certain quorum had signed a set of updates before installing. Since, at present, compromising the security of the single machine which is building the updates would allow an attacker to issue trojaned "updates" to a large number of machines, distributing the building process would certainly be advantageous.

Ideally, one would hope that some day the convoluted methods used here will be unnecessary. Software installed from the ports tree, and some other operating systems, have the advantage of being completely packagized; this makes it easy to install or remove specific packages, and consequently makes it trivial to update everything on the system. If FreeBSD were split into such independent packages, a wide range of problems

would be simplified; however, performing such a task would most likely require reworking the entire build system, and it seems likely that development will never stop for long enough to make such an effort possible, even if someone could be found with the necessary capability and time to perform such a task.

9 Acknowledgements

The author would like to thank Chad David and Terry Lambert for their assistance in explaining the FreeBSD build process; Nathan Sidwell for his assistance with gcc; and Graham Percival for replacing a dead hard drive, repairing a broken filesystem on a working hard drive, and otherwise helping to maintain the author's FreeBSD box while he was 4700 miles away.

The author would also like to acknowledge support from the Commonwealth Scholarship Commission, which is funding his studies at Oxford University.

10 Availability

The client (update installing) and server (update building) code is available under an open source license from

`http://www.daemonology.net/freebsd-update/`

The client code is also available in the FreeBSD ports tree.

References

[An02] M. Anuzis, *Incident Analysis of a Compromised OpenBSD 3.0 Honeypot*, `http://www.anuzisnetworking.com/whitepapers/obsd30/` (2002).

[Bi02] *FreeBSD Binary Updater Project (binup)*, `http://www.freebsd.org/projects/updater.html` (2002).

[CE03] *CERT Advisory CA-2003-10 Integer overflow in Sun RPC XDR library routines*, `http://www.cert.org/advisories/CA-2003-10.html` (2003).

[Ga03] Gerardo Santana Gómez Garrido, *Binary patches for OpenBSD*, `http://www.openbsd.org.mx/~santana/binpatch.html` (2003).

[Ho02] S. Holcroft, *Incident Analysis of a Compromised RedHat Linux 6.2 Honeypot*, `http://www.holcroft.org/honeypot/Incident/sholcroft-4.1-2002.html` (2002).

[IS03] InstallShield, *InstallShield - Update Service*, `http://www.installshield.com/isus/` (2003).

[La03] D. Langille, Personal email (23 June 2003).

[NP03] *The NEO Project*, `http://www.theneoproject.com/` (2003).

[Pe03] C. Percival, *Naive Differences of Executable Code*, `http://www.daemonology.net/bsdiff/` (2003).

[Po02] J. Polstra, *CVSup*, `http://www.cvsup.org/` (2002).

[Pr03] M. Prettejohn, Personal email (27 May 2003).

[PST03] Puget Sound Technology, *Binary Updates for NetBSD*, `http://pugetsoundtechnology.com/services/netbsd/updates/` (2003).

[Ri92] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321 (1992).

[Ro96] M.J.B. Robshaw, *On Recent Results for MD2, MD4, and MD5*, RSA Laboratories Bulletin, November 1996.

[Si03] N. Sidwell, Personal email (14 Feb 2003).

[SPW02] S. Staniford, V. Paxson, and N. Weaver, *How to Own the Internet in Your Spare Time*, Proceedings of the 11th USENIX Security Symposium (2002).

[SA02] *FreeBSD Security Advisories SA-01:40, SA-01:42, SA-01:48, SA-01:49, SA-01:51, SA-01:52, SA-01:53, SA-01:55, SA-01:56, SA-01:57, SA-01:58, SA-01:59, SA-01:62, SA-01:63, SA-02:08, SA-02:13, and SA-02:25*, `http://www.freebsd.org/security` (2002).

Building a High-performance Computing Cluster Using FreeBSD

Brooks Davis, Michael AuYeung, Gary Green, Craig Lee
The Aerospace Corporation
El Segundo, CA
{brooks,lee,mauyeung}@aero.org, Gary.B.Green@notes.aero.org

Abstract

In this paper we discuss the design and implementation of Fellowship, a 300+ CPU, general use computing cluster based on FreeBSD. We address the design features including configuration management, network booting of nodes, and scheduling which make this cluster unique and how FreeBSD helped (and hindered) our efforts to make this design a reality.

1 Introduction

For most of the last decade the primary thrust of high performance computing (HPC) development has been in the direction of commodity clusters, commonly known as Beowulf clusters [Becker]. These clusters combine commercial off-the-shelf hardware to create systems which rival or exceed the performance of traditional supercomputers in many applications while costing as much as a factor of ten less. Not all applications are suitable for clusters, but a signification portion of interesting scientific applications can be adapted to them.

In 2001, driven by a number of separate users with supercomputing needs, The Aerospace Corporation (a non-profit, federally funded research and development center) decided to build a corporate computing cluster (eventually named Fellowship [1]) as an alternative to continuing to buy small clusters and SMP systems on an ad-hoc basis. This decision was motivated by a desire to use computing resources more efficiently as well as reducing administrative costs. The diverse set of user requirements in our environment led us to a design which differs significantly from most clusters we have seen elsewhere. This is especially true in the areas of operating system choice (FreeBSD) and configuration management (fully network booted nodes).

Fellowship is operational and being used to solve significant real world problems. Our best benchmark run so far has achieved 183 GFlops of floating point performance which would place us in the top 100 on the 2002 TOP500 clusters list.

In this paper, we first give an overview of the cluster's configuration. We cover the basic hardware and software, the physical and logical layout of the systems, and basic operations. Second, we discuss in detail the major design issues we faced when designing the cluster, how we chose to resolve them, and discuss the results of these choices. In this section, we focus particularly on issues related to our use of FreeBSD. Third, we discuss lessons learned as well as lessons we wish the wider parallel computing community would learn. Fourth, we talk about future directions for the community to explore either in incremental improvements or researching new paradigms in cluster computing. Finally, we sum up where we are and where we are going. Table 2 contains a listing of URLs for many of the projects or products we mention.

2 Fellowship Overview

The basic logical and physical layout of Fellowship is similar to many clusters. There are three core systems, 151 dual-processor nodes, a network switch, and assorted remote management hardware. All nodes and servers run FreeBSD, currently 4.8-STABLE. The core systems and remote manage-

[1] Originally, it was suggested that we name the cluster Frodo, but the need for a name which would provide us with multiple names for core equipment drove us to use Fellowship as the name of the cluster.

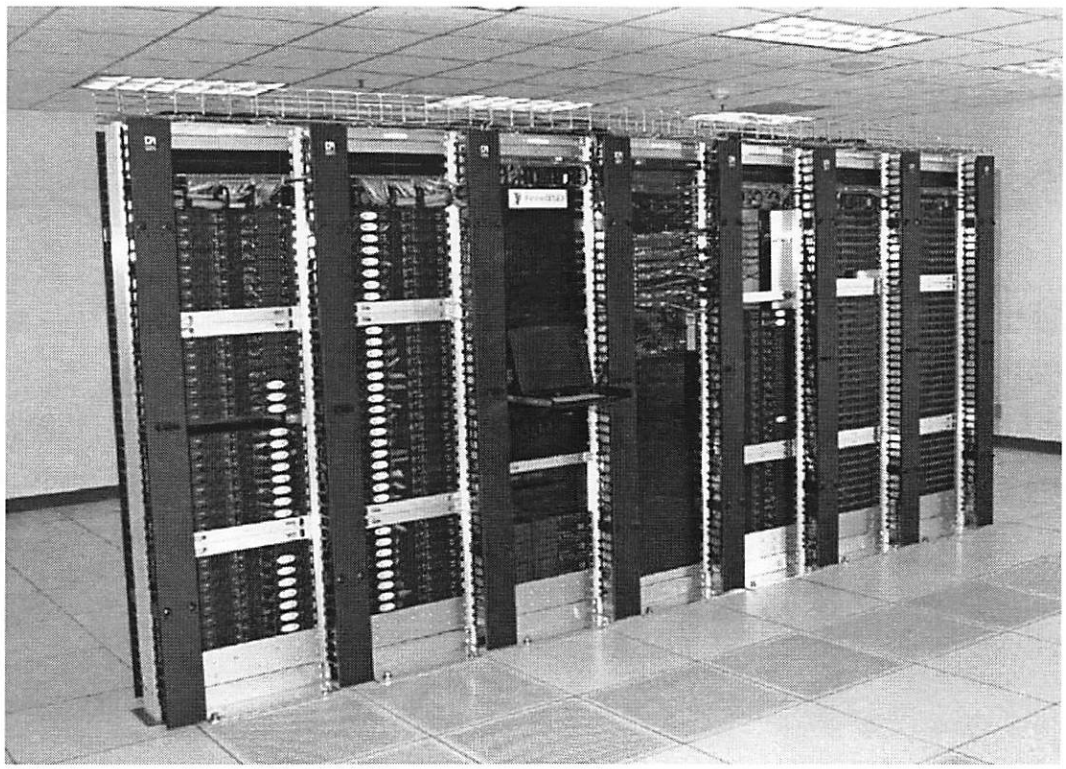

Figure 1: Fellowship Circa April 2003

ment hardware sit on the Aerospace corporate network. The nodes and core systems share a private, non-routed network (10.5/16). This equipment is mounted in a row of seven-foot tall, two-post racks residing in the underground data center at Aerospace headquarters in El Segundo, California. Figure 1 shows Fellowship in April 2003. The layout of the node racks is shown in Figure 2.

The core systems are a user or shell server, a data server which serves NFS shared scratch space and does backups, and a management server which runs the scheduler, serves NIS, and manages the nodes. The user server, `fellowship`, is the gateway through which users access the cluster. Users log into it and launch jobs from there. Home directories are stored on `fellowship` and exported via NFS to the nodes. The data server, `gamgee`, hosts 271 GB of shared scratch space for use by users during computations. It also runs a MySQL database for results storage and AMANDA for backups of key cluster systems. The management server, `frodo`, performs a wide variety of tasks. These include exporting account information via NIS, network booting the nodes, and scheduling user jobs.

The nodes are dual CPU x86 systems ranging from 1 GHz Pentium IIIs to 2.4GHz Xeons with 1GB of RAM installed. Table 1 gives a complete breakdown of CPU types used in Fellowship. All Pentium III nodes were purchased with 40GB IDE disks. The Xeon nodes were purchased with 80GB disks and Pentium III disks are being replaced with 80GB disks as they fail. The nodes are connected via Gigabit Ethernet through a Cisco Catalyst 6513 switch. The Pentium III systems are Tyan Thunder LE (1GHz systems) and Tyan Thunder LE-T with 3Com Gigabit Ethernet adapters installed in their expansion slots. They are mounted in 14" deep rackmount cases and were integrated by iXsystems. The Xeon systems are Intel 1U server platforms with dual on board Gigabit Ethernet interfaces. We purchased them from Iron Systems.

CPU Type	*Nodes*	*CPUs*
Pentium III 1GHz	8	16
Pentium III 1.26GHz	40	80
Pentium III 1.4GHz	39	78
Xeon (P4) 2.4GHz	64	128
Total	151	302

Table 1: CPUs in Fellowship nodes.

Unit	*Contents*
45	filler panel (*empty*)
44	filler panel (*empty*)
43	filler panel (*empty*)
42	48-port terminal server (r01ts: 10.5.1.0)
41	Cable management
40	
39	node (r01n32: 10.5.1.32)
38	node (r01n31: 10.5.1.31)
37	node (r01n30: 10.5.1.30)
36	node (r01n29: 10.5.1.29)
35	node (r01n28: 10.5.1.28)
34	node (r01n27: 10.5.1.27)
33	node (r01n26: 10.5.1.26)
32	node (r01n25: 10.5.1.25)
31	Baytech RPC4 8-port power controller
30	Baytech RPC4 8-port power controller
29	node (r01n24: 10.5.1.24)
28	node (r01n23: 10.5.1.23)
27	node (r01n22: 10.5.1.22)
26	node (r01n21: 10.5.1.21)
25	node (r01n20: 10.5.1.20)
24	node (r01n19: 10.5.1.19)
23	node (r01n18: 10.5.1.18)
22	node (r01n17: 10.5.1.17)
21	node (r01n16: 10.5.1.16)
20	node (r01n15: 10.5.1.15)
19	node (r01n14: 10.5.1.14)
18	node (r01n13: 10.5.1.13)
17	node (r01n12: 10.5.1.12)
16	node (r01n11: 10.5.1.11)
15	node (r01n10: 10.5.1.10)
14	node (r01n09: 10.5.1.9)
13	Baytech RPC4 8-port power controller
12	Baytech RPC4 8-port power controller
11	node (r01n08: 10.5.1.8)
10	node (r01n07: 10.5.1.7)
9	node (r01n06: 10.5.1.6)
8	node (r01n05: 10.5.1.5)
7	node (r01n04: 10.5.1.4)
6	node (r01n03: 10.5.1.3)
5	node (r01n02: 10.5.1.2)
4	node (r01n01: 10.5.1.1)
3	
2	5 110V 20A Circuits
1	

Figure 2: Layout of Node Rack 1

Although the nodes have disks, we network boot them using PXE support on their network interfaces with `frodo` providing DHCP, TFTP, NFS root disk, and NIS user accounts. On boot, the disks are automatically checked to verify that they are properly partitioned for our environment. If they are not, they are automatically repartitioned. This means no manual configuration of nodes is required beyond determining their MAC address when they are installed.

Local control of cluster machines is made possible through a KVM-switch connected to a 1U rackmount LCD keyboard, monitor, and track pad. Remote access is available through Cyclades TS-series terminal servers. All nodes and servers as well as networking gear are connected to these terminal servers and console redirection is enabled on all FreeBSD machines. We have BIOS console redirection enabled on the Xeon systems, but not on the Pentium III systems as a bug tends to cause them to hang, even at very low baud rates. In addition to console access, everything except the terminal servers and the switch are connected to BayTech RPC4-15 serial remote power controllers. This allows us to remotely reboot virtually any part of the system by connecting to the power controller via the appropriate terminal server.

On top of this infrastructure, access to nodes is controlled by Sun Grid Engine (SGE), a scheduler implementing a superset of the POSIX Batch Environment Services specification. SGE allows users to submit both interactive and batch job scripts to be run on one or more processors. Users are free to use the processors they are allocated in any reasonable manner. They can run multiple unrelated processes or single massively parallel jobs.

To facilitate use of Fellowship, we provide a basic Unix programming environment, plus the parallel programming toolkits, and commercial parallel applications. For parallel programming toolkits we provide Parallel Virtual Machine and the MPICH and LAM implementations of the Message Passing Interface [MPI]. Currently, our sole commercial parallel application is Grid Mathematica for which we were the launch customer.

3 Design Issues

One of the biggest challenges in building Fellowship was our diverse user base. Among the users at the initial meetings to discuss cluster architecture, we had users with loosely coupled and tightly coupled applications, data intensive and non-data intensive applications, and users doing work ranging from daily production runs to high performance computing research. This diversity of users and applications led to the compromise that is our current

Resource	*URL*
Big Sister	`http://bigsister.graeff.com/`
BProc	`http://bproc.sourceforge.net/`
Diskmark	`http://people.freebsd.org/~brooks/diskmark/`
Diskprep (enhanced)	`http://people.freebsd.org/~brooks/diskprep/`
Diskprep (original)	`http://people.freebsd.org/~imp/diskprep.pl`
DQS	`http://www.scri.fsu.edu/~pasko/dqs.html`
EmuLab	`http://www.emulab.net/`
FreeBSD	`http://www.FreeBSD.org/`
Ganglia Cluster Monitor	`http://ganglia.sourceforge.net/`
GEOM Overview	`http://phk.freebsd.dk/geom/overview.txt`
Global File System (GFS)	`http://www.sistina.com/products\_gfs.htm`
Grid Mathematica	`http://www.wolfram.com/products/gridmathematica/`
LAM-MPI	`http://www.lam-mpi.org/`
LinuxBIOS	`http://www.linuxbios.org/`
LSF	`http://www.platform.com/products/wm/LSF/`
Maui Scheduler	`http://www.supercluster.org/maui/`
Myrinet	`http://www.myri.com/myrinet/`
MPICH	`http://www-unix.mcs.anl.gov/mpi/mpich/`
Nagios	`http://www.nagios.org/`
OpenPBS	`http://www.openpbs.org/`
Parallel Virtual Machine	`http://www.csm.ornl.gov/pvm/`
Rocks Cluster Distribution	`http://www.rocksclusters.org/`
Scalable OpenPBS	`http://www.supercluster.org/projects/pbs/`
Sun Grid Engine (SGE)	`http://gridengine.sunsource.net/`
Clusters @ TOP500	`http://clusters.top500.org/`

Table 2: Resources for Clusters

design. In this section we highlight the major design decisions we made while building Fellowship.

3.1 Operating System

The first major design decision any cluster faces is usually the choice of operating system. By far, the most popular choice is some Linux distribution. Certainly Linux is the path of least resistance and most people assume that, if it is a cluster, it runs Linux. In fact a cluster can run almost any operating system. Clusters exist running Solaris [SciClone], HP-UX, AIX, MacOS X, FreeBSD [Jeong, Schweitzer], and even Windows. ASCI Blue Mountain is actually a cluster of 48 128-CPU SGI systems running Irix [SGI].

For an organization with no operating system bias and straight-forward computing requirements, running Linux is the path of least resistance due to free clustering toolkits such as NPACI's Rocks Cluster Distribution. In other situations, operating system choice is more complicated. Important factors to consider include chosen hardware platform, existence of experienced local system administration staff, availability of needed applications, easy of maintenance, system performance, and the importance of the ability to modify the operating system.

For a variety of reasons, we chose FreeBSD for Fellowship. The most pragmatic reason for doing so is the excellent out of the box support for diskless systems which was easily modifiable to support our nodes network booting model. This part has worked out very well.

Additionally, the chief Fellowship architect uses FreeBSD almost exclusively and is a FreeBSD committer. This meant we had more FreeBSD experience than Linux experience and that we could push some of our more general changes back into FreeBSD to simplify operating system upgrades. In practice, our attempts to push changes back into the base operating system have met with mixed success. We have merged a few small changes, but the generally applicable portion of our diskless boot script changes have not been merged due to lack of time to sort out

conflicting changes to the main source tree.

The ports collection was also a major advantage of using FreeBSD. It has allowed us to install and maintain user-requested software quickly and easily. In some cases, existing ports were not flexible enough for our needs, but for most applications, it works well. The availability of Linux emulation meant we did not give up much in the way of application compatibility. We have successfully run Grid Mathematica on the cluster after following the Mathematica installation documentation in the FreeBSD Handbook.

The disadvantages of FreeBSD for our purposes are immature SMP and threading support, and an widely held view within the high performance computing community that if it isn't a commercial supercomputer, it must be a Linux system. SMP support has not been a major issue for our users to date. Most of our jobs are compute-bound so the poor SMP performance under heavy IO is a moot problem. Threading has been more of an issue. We have users who would like to use threading for SMP scaling. We expect this situation to improve when we migrate to FreeBSD 5.x.

The Linux focus of the HPC community has caused us some problems. In particular, many pieces of software either lack a FreeBSD port, or only have a poorly tested one which does not actually work. Additionally, there is a distinct shortage of compiler support for modern versions of FORTRAN.

3.2 Hardware Architecture

The choice of hardware architecture is generally made in conjunction with the operating system as the two interact with each other. Today, most clusters are based on Intel or AMD x86 CPUs, but many other choices are available. 64-bit SPARC and Alpha clusters are fairly common, and clusters based on Apple's XServe platform are popular in Macintosh shops. The major issues to consider are price, performance, power consumption, and operating system compatibility. For instance, Intel's Itanium2 has excellent performance, but is expensive and power hungry as well as suffering from immature operating system support. In general, x86 based systems are currently the path of least resistance given the lack of a conflicting operating system requirement.

When we were selecting a hardware architecture in 2001, the major contenders were Alpha and Intel or AMD based x86 systems. We quickly discarded Alpha from consideration because of previous experiences with overheating problems on a small Aerospace Alpha cluster. Alphas also no longer have the kind of performance lead they enjoyed in the late 1990's. We looked at both Pentium III and Athlon-based systems, but decided that while the performance characteristics and prices did not vary significantly, power consumption was too problematic on the Athlon systems.

Over the life of Fellowship, we have investigated other types of nodes including newer Athlon based systems, the Xeon systems we purchased in this year's expansion, Apple XServes, and now AMD Opteron systems. Athlons have failed to match the power/performance ratios of Intel systems. Similarly, XServes are attractive, but offer sub-par performance and little improvement in power consumption in addition to being an incompatible architecture. We will not make a decision until we know what the hardware market landscape looks like late this year, but preliminary reports seem to indicate that the amd64 port of FreeBSD will allow us to explore using systems with much larger system memories while retaining x86 compatibility for users who do not want to think about which machines they are running on.

3.3 Node Architecture

Most of the decisions about node hardware will derive from the selection of hardware architecture, cluster form factor, and network interface. The biggest of the remaining choices is single or multi-processor systems. Single processor systems have better CPU utilization due to a lack of contention for RAM, disk, and network access. Multi-processor systems can allow hybrid applications to share data directly, decreasing their communication overhead. Additionally, multi-processor systems tend to have higher performance external interfaces then single processor system.

Other choices are processor speed, RAM, and disk space. We have found that aiming for the knee of the price curve has served us well, since no single user dominates our decisions. In other environments, top of the line processors, very large disks, or large amounts of RAM may be justified despite the exponential increase in cost.

CPU	2 x Pentium III 1GHz
Network Interface	3Com 3C996B-T
RAM	1GB
Disk	40GB 7200RPM IDE

Table 3: Configuration of first Fellowship nodes.

CPU	2 x Xeon 2.4GHz
Network Interface	On board gigabit
RAM	2GB
Disk	80GB 7200RPM IDE

Table 4: Configuration of latest Fellowship nodes.

For Fellowship, we chose dual CPU systems. We were motivated by a desire to do research on code that takes advantage of SMP systems in a cluster, higher density than single processor systems, and the fact that the 64-bit PCI slots we needed for Gigabit Ethernet were not available on single CPU systems. As a result of our focus on the knee of the price curve, we have bought slightly below the performance peak on processor speed, with 2-4 sticks of smaller then maximum RAM, and disks in the same size range as mid-range desktops. This resulted in the initial configuration shown in Table 3. The most recent node configuration is shown in Table 4.

3.4 Network Interconnects

Like hardware architecture, the selection of network interfaces is a matter of choosing the appropriate point in the trade space between price and performance. Performance is generally characterized by bandwidth and latency. The right interface for a given cluster depends significantly on the jobs it will run. For loosely coupled jobs with small input and output datasets, little bandwidth is required and 100Mbps Ethernet is the obvious choice. For other, tightly coupled jobs, Myrinet with its low latency and 2 Gbps+2 Gbps bandwidth is the right solution. Other interfaces, such as upcoming InfiniBand products, provide alternatives for high speed interfaces.

The choice of Gigabit Ethernet for Fellowship's interconnect represents a compromise between the cheaper 100 Mbps Ethernet our loosely coupled applications would prefer (allowing us to buy more nodes) and Myrinet. We plan to improve the efficiently of our network by upgrading to use JumboFrames (9000 byte MTUs) in the near future. When we started building Fellowship, Gigabit Ethernet was about one-third of the cost of each node whereas Myrinet would have more than doubled our costs. Looking to our expansion next year, Gigabit Ethernet is standard on the motherboard, and with the large switches our cluster requires, the cost per port is less then 20% higher then 100Mbps Ethernet. We are considering the idea of creating sub-clusters within Fellowship with faster network interfaces such as Myrinet.

3.5 Addressing and Naming Schemes

There are three basic approaches to allocating IP addresses in a cluster. For small clusters, many architects simply put all the machines on an existing network. This has the advantage that no additional routing is needed for the nodes to talk to arbitrary external data sources. The disadvantage is that it typically means the IP-addresses do not correspond to physical objects so it is hard to distinguish machines. Additionally, not subnetting the cluster can make it too easy for inter-node communication to impact the rest of the network. The other two approaches involve placing nodes on their own subnet, either with public or private [RFC1918] addresses. Using public addresses has the advantage that with appropriate routers, cluster nodes can exchange data with arbitrary external data sources. On a subnet, IP-addresses can be mnemonic to help administrators remember which machine a particular address belongs to. The main disadvantage of using public addresses is that address space is becoming increasingly scarce and large allocations are difficult or expensive to obtain. The use of private addresses eliminates this pressure by allowing the use of 2^{24} addresses in the 10/8 address space. This allows useful mnemonic naming schemes without any pressures to use addresses efficiently. The disadvantage is that nodes cannot reach external data sources directly. If all they need to do is access HTTP or FTP servers, a proxy can be used, but many grid computing tools assume that all machines in a computation are on fully routed networks.

On Fellowship we chose to use the 10.5/16 private network. We chose this approach because we needed our own subnet to avoid consuming other networks resources and we would have needed at least a /23 allocation, which was not available at the time. Within our network 10.5.0/24 is reserved for core equipment. 10.5.255/24 is available for temporary

DHCP allocation to allow devices to acquire a network address before they have their MAC address recorded in the DHCP config file. The $10.5.X/24$ blocks are allocated to node racks numbered from 1. Originally, $10.5.X.0$ was the terminal server for that rack and $10.5.X.Y$ $(0 < Y < 255)$ corresponds to node Y within that rack. We have since moved the terminal servers onto the corporate network because they will not support JumboFrames. This allocation scheme would not be possible with public addresses due to address allocation authority requirements.

Choosing host names within a cluster is another issue faced by a cluster architect. The usual rules of host naming [RFC1178] apply to naming core servers. However, unless the cluster is very small and likely to remain so, a numerical naming scheme such as `node00`, `node01`, etc. is likely to be a better idea then trying to come up with a naming scheme that can handle hundreds of unique machines.

For Fellowship, we choose to name our core machines after members of The Fellowship of the Ring [Tolkien]. At some point we may run out of names and need to start using other characters from The Lord of the Rings, but the theme should easily hold for the core systems. We choose to name nodes after their host rack and their position within that rack. Nodes are numbered from the bottom (because we fill the racks from the bottom). Thus each node's name looks like `r##n##` with the first node in rack 1 being `r01n01`. Terminal servers were originally named `r##ts`, but have since been changed to `gimli-r##` with `gimli` being the terminal server for the core systems. The nice things about naming devices in the node racks this way is that conversion between IP-addresses and host names can be accomplished with a simple regular expression.

Domain names add a slight complication to the naming process. It is often useful to make the cluster into its own DNS zone. With Fellowship, all external systems reside within the `aero.org` zone and nodes reside within an internal use only `fellow.aero.org` zone. The disadvantage of this is that some software prefers that hosts are within the same zone.

3.6 Core Servers and Services

On Fellowship, we refer to all the equipment other then the nodes and the remote administration hardware as core servers. On many clusters, a single core server suffices to provide all necessary core services. In fact, some clusters simply pick a node to be the nominal head of the cluster. Some large clusters provide multiple front ends, with load balancing and failover support to improve uptime.

Core services are those services which need to be available for users to utilize the cluster. At a minimum, users need accounts and home directories. They also need a way to configure their jobs and get them to the nodes. The usual way to provide these services is to provide shared home and application directories, usually via NFS and use a directory service such as NIS to distribute account information. Other core services a cluster architect might choose to include are batch schedulers, databases for results storage, and access to archival storage resources. The number of ways to allocate core servers to core services is practically unlimited.

Fellowship has three core servers: the data server, the user server, and the management server. All of these servers are currently 1GHz Pentium III systems with SCSI RAID5 arrays. The data server, `gamgee`, serves a 250GB shared scratch volume via NFS, runs a MySQL database for users to store results in, and does nightly backups to a 20 tape library using AMANDA. We are in the process of upgrading the scratch portion of the data server to a dual Xeon box containing 2.8TB of IDE RAID. Backups and databases will remain on `gamgee`. The user server, `fellowship`, serves NFS home directories and gives the users a place to log in to compile and run applications. The management server, `frodo`, hosts the scheduler, NIS, and our shared application hierarchy mounted at `/usr/aero`. Additionally, the management server uses DHCP, TFTP, and NFS to netboot the nodes. We are in the process of upgrading `fellowship` and `frodo` to dual 2.4GHz Xeons with 285GB of SCSI RAID5 storage each, doubling their previous capacity.

These services were isolated from each other for performance reasons. In our model, hitting the shared scratch space does not slow down ordinary compiles and compiling does not slow down scratch space access. We discovered that, separation of services does work, but it comes at the cost of increased fragility because the systems are interdependent, and when one fails, they all have problems. We have devised solutions to these problems, but this sort of division of services should be carefully planned and would generally benefit from redundancy when feasible. Given unlimited funds, we would probably move most NFS service to an appliance type device

such as a NetApp file server.

3.7 Node Configuration Management

Since nodes generally outnumber everything else on the system, efficient configuration management is essential. Many systems install an operating system on each node and configure the node-specific portion of the installation manually. Other systems network boot the nodes using Etherboot, PXE or LinuxBIOS. The key is good use of centralization and automation. We have seen many clusters where the nodes are never updated without dire need because the architect made poor choices that made upgrading nodes impractical.

Node configuration management is probably the most unique part of Fellowship's architecture. We start with the basic FreeBSD diskless boot process [Perlstein]. We then use the diskless remount support to mount `/etc` as `/conf/base/etc` and override ssh keys on the nodes. For many applications, this configuration would be sufficient. However, we have applications which require significant amounts of local scratch space. As such, each node contains a disk. The usual way of handling such disks would be to manually create appropriate directory structures on the disk when the system was first installed and then let the nodes mount and fsck the disks each time they were booted. We deemed this impractical because nodes are usually installed in large groups. Additionally, we wanted the ability to reconfigure the disk along with the operating system. Instead of manual disk configuration, we created a program (`diskmark`) which uses an invalid entry in the MBR partition table to store a magic number and version representing the current partitioning scheme. At boot we use a script which executes before the body of `rc.diskless2` to examine this entry to see if the current layout of the disk is the required one. If it is not, the diskless scripts automatically use Warner Losh's `diskprep` script to initialize the disk according to our requirements.

With this configuration, adding nodes is very easy. The basic procedure is to bolt them into the rack, hook them up, and turn them on. We then obtain their MAC address from the switch's management console and add it to the DHCP configuration so each node is assigned a well-known IP address. After running a script to tell the scheduler about the nodes and rebooting them, they are ready for use.

Maintenance of the netboot image is handed by chrooting to the root of the installation and following standard procedures to upgrade the operating system and ports as needed. For operating system upgrades, we copy the entire root to a new location, upgrade it, and test a few nodes before modifying the DHCP configuration for all nodes and rebooting them to use the new root. We install software available through the ports collection via the standard process and manage it with portupgrade. Software which is not available in the ports collection is installed in the separate `/usr/aero` hierarchy.

One part of network booting Fellowship's nodes that has not worked out as planned is BIOS support for PXE. PXE is a standard feature on server-class motherboards, but seems to be poorly tested by manufacturers. More than once, our vendor had to go back to the motherboard manufacture to have them create a new BIOS to fix a PXE problem. We have found PXE to be somewhat unreliable on nearly all platforms, occasionally failing to boot from the network for no apparent reason and then falling back to the disk which is not configured to boot. Some of these problems appear to be caused by interactions with network switches, particularly Cisco switches. Recently, we have been working on an enhanced version of `diskprep` which will allow us to create a FreeDOS partition that will automatically reboot the machine, giving it infinite retries at PXE booting.

3.8 Job Scheduling

Job scheduling is potentially one of the most complex and contentious issues faced by a cluster architect. The major scheduling options are running without any scheduling, manual scheduling, batch queuing, and domain specific scheduling.

In small environments with users who have compatible goals, not having a scheduler and just letting users run what they want when they want or communicating with each other out of band to reserve resources as necessary can be a good solution. It has very little administrative overhead, and in many cases, it just works.

With large clusters, some form of scheduling is usually required. Even if users do not have conflicting goals, it's difficult to try to figure out which nodes to run on when there are tens or hundreds available. Additionally, many clusters have multiple purposes

Mountpoint	*Source*
/	frodo:/nodedata/roots/freebsd/4.8-STABLE
/conf/base/etc	frodo:/nodedata/roots/freebsd/4.8-STABLE/etc
/etc	mfs
/usr/aero	frodo:/nodedata/usr.aero
/tmp	/dev/ad0s2a
/var	/dev/ad0s2d
/home	fellowship:/home
/scratch	gamgee:/scratch
/db	gamgee:/db
/dev	mfs

Table 5: Sample node (r01n01 aka 10.5.1.1) mount structure

that must be balanced. In many environments, a batch queuing system is the answer. A number exist, including OpenPBS, PBSPro, Sun Grid Engine (SGE), LSF, NQS, and DQS. These systems typically include a scheduler, but many of them also support running the Maui backfill scheduler on top of them. OpenPBS and SGE are freely available open source applications and are the most popular options for cluster scheduling.

For some applications, batch queuing is not a good answer. This is usually either because the application requires that too many jobs for most batch queuing systems to keep up or because the runtime of jobs is too variable to be useful. For instance, we have heard of one computational biology application which runs through tens of thousands of test cases a day where most take a few seconds, but some may take minutes, hours, or days to complete. In these situations, a domain specific scheduler is often necessary. A common solution is to store cases in a database and have applications on each node that query the database for a work unit, process it, store the result in the database, and repeat.

On Fellowship, we have a wide mix of applications ranging from trivially scheduleable tasks to applications with unknown run times. Our current strategy is to implement batch queuing with a long-term goal of discovering a way to handle very long running applications. We initially intended to run the popular OpenPBS scheduler because it already had a port to FreeBSD and it is open source. Unfortunately, we found that OpenPBS had major stability problems under FreeBSD (and, by many accounts, most other operating systems) [2]. About the time we were ready to give up on OpenPBS, Sun released SGE as open source. FreeBSD was not supported initially, but we were able to successfully complete a port based on some patches posted to the mailing lists. We have since contributed that port back to the main SGE source tree.

[2] A recent fork called Scalable OpenPBS may eventually remedy these issues.

3.9 Security Considerations

For most clusters, we feel that treating the cluster as a single system is the most practical approach to security. Thus for nodes which are not routed to the Internet like those on Fellowship, all exploits on nodes should be considered local. What this means to a given cluster's security policy is a local issue. For systems with routed nodes, management gets more complicated, since each node becomes a source of potential remote vulnerability. In this case it may be necessary to take action to protect successful attacks on nodes from being leveraged into full system access. In such situations, encouraging the use of encrypted protocols within the cluster may be desirable, but the performance impact should be kept firmly in mind.

The major exception to this situation are clusters that require multi-level security. We have some interest in the issues in such a system, but at this point have not done any serious investigation.

We have chosen to concentrate on protecting Fellowship from the network at large. This primarily consists of keeping the core systems up to date and requiring that all communications be via encrypted protocols such as SSH. Internally we encourage the use of SSH for connecting to nodes, but do allow RSH connections. Our Sun Grid Engine install uses a PKI-based user authentication scheme. We discovered this is necessary because SGE's default privilege

model is actually worse than RSH in that it does not even require the dubious protection of a lower port. Inter-node communications are unencrypted for performance reasons.

3.10 System Monitoring

The smooth operation of a cluster can be aided by proper use of system monitoring tools. Most common monitoring tools such as Nagios and Big Sister are applicable to cluster use. The one kind of monitoring tool that does not work well with clusters is the sort that sends regular e-mail reports for each node. Even a few nodes will generate more reports then most admins have time to read. In addition to standard monitoring tools, there exist cluster specific tools such as the Ganglia Cluster Monitor. Most schedulers also contain monitoring functionality.

On Fellowship we are currently running the Ganglia Cluster Monitoring system and the standard FreeBSD periodic scripts on core systems. Ganglia was ported to FreeBSD previously, but we have created FreeBSD ports which make it easier to install and make its installation more BSD-like. A major advantage of Ganglia is that no configuration is required to add nodes. They are automatically discovered via multicast. We have also considered using Nagios to monitor nodes, but have not yet successfully deployed it. Monitoring is an area we need to improve on Fellowship. We have had disks fail after a reboot without anyone noticing, because the default FreeBSD diskless behavior causes it to boot anyway. It was nice that the nodes kept working, but we were surprised to find some machines had small memory based `/tmp` directories instead of 36GB+ disk based ones.

3.11 Physical System Management

At some point in time, every system administrator finds that they need to access the console of a machine or power cycle it. With just a few machines, installing monitors on each machine or installing a KVM switch for all machines and flipping power switches manually is a reasonable option. For a large cluster, installing serial terminal servers to allow remote access to consoles and remote power controllers may be advisable.

In Fellowship's architecture, we place a strong emphasis on remote management. The cluster is housed in our controlled access data center, which makes physical access cumbersome. Additionally, the chief architect and administrator lives 1000 miles from the data center, making direct access even more difficult. As a result, we have configured all computers to provide remote console access via terminal servers and have provided their power through remote power controllers. This allows us to reliably reboot systems at will, which greatly aids recovery and remote diagnosis of faults. Not all problems can be solved this way, but many can. We were able to diagnose a reboot caused by running out of network resources, but not a crash caused by a RAID controller that died. We have had mixed results with BIOS console access. On the Intel Xeon systems it works well, but the Tyan Pentium III motherboards tend to hang on boot if BIOS console redirection is enabled. In both cases we are able to access FreeBSD's console, which has proven useful.

3.12 Form Factor

The choice of system form factor is generally a choice between desktop systems on shelves versus rack mounted servers. Shelves of desktops are common for small clusters as they are usually cheaper and less likely to have cooling problems. Their disadvantages include the fact that they take up more space, the lack of cable management leading to more difficult maintenance, and generally poor aesthetics. Additionally, most such systems violate seismic safety regulations.

Rack mounted systems are typically more expensive due to components which are produced in much lower volumes as well as higher margins in the server market. Additionally, racks or cabinets cost more then cheap metal shelves. In return for this added expense, rackmount systems deliver higher density, integrated cable management, and, usually, improved aesthetics.

Higher density is a two-edged sword. Low-end cases are often poorly designed and inadequately tested, resulting in overheating due to cramped quarters and badly routed cables. Additionally, a single rack can generate an amazing amount of heat. We estimate there is a 20-30 degree (F) difference between the front and back of Fellowship's racks of Xeons despite being in a well air conditioned underground data center. Those racks have a peak power consumption of over 6000W each.

A minor sub-issue related to rackmount systems is cabinets vs. open, telco style racks. Cabinets look more polished and can theoretically be moved around. Their disadvantages are increased cost, lack of space making them hard to work in, and being prone to overheating due to restricted airflow. Telco racks do not look as neat and are generally bolted to the floor, but they allow easy access to cables and unrestricted airflow. In our case, we use vertical cable management with doors which makes Fellowship look fairly neat without requiring cabinets.

The projected size of Fellowship drove us to a rackmount configuration immediately. We planned from the start to eventually have at least 300 CPUs, which is pushing reasonable bounds with shelves. The only thing that has not gone well with our racks is that we chose six inch wide vertical cable management, which gets cramped at times. We plan to use ten inch wide vertical cable management when we expand to a second row of racks next fiscal year.

4 Lessons Learned

The biggest lesson we have learned is that hardware attrition is a real issue. While we have not seen many hard-to-track instability problems, we have lost at least one machine nearly every time we have had a building-wide power outage, scheduled or unscheduled. As a result, we have learned that it is important to have a vendor who will repair failed or failing systems quickly. The fact that nodes fail more frequently then we had initially expected also means that neat cabling is more crucial then we first thought. To save money in the initial deployment, we ran cables directly from the switch to the nodes. This means we have a lot of slack cable in the cable management, which makes removing and reinstalling nodes difficult. When we expand the cluster to a second row of racks next year, we plan to switch to having patch panels at the top of each rack connecting to panels beside the switch.

We have also learned that while most HPC software works fine on FreeBSD, the high performance computing community strongly believes the world is a Linux box. It is often difficult to determine if a problem is due to inadequate testing of the code under FreeBSD or something else. We hope that more FreeBSD users will consider clustering with FreeBSD.

System automation is even more important then we first assumed. For example, shutting down the system for a power outage can be done remotely, but currently it requires logging in to all 20 remote power controllers. We are currently working on automating this as well as adding automatic shutdown of nodes in the event of external power loss.

5 Future Directions & Conclusions

Currently Fellowship is working well, but there are still improvements to be made, particularly in the areas of automation and scheduling.

We have planned for an evolving system, but we have not actually got to the stage of replacing old hardware so we do not know how that is going to work in practice. Clearly, at some point, nodes will be wasting more power then they are worth, but we do not know what that point is. A measure of FLOPS/Watt will be helpful in determining this. We also do not know if systems will start failing en masse in the future or if they will die slowly over a long period of time.

Other directions we need to pursue are in the area of scheduling. We need to better handle job models that do not fit well within the batch paradigm where users have a good idea how long their jobs will run. Some of our users have jobs that will run for weeks or months at a time, so this is a pressing concern. We are currently pursuing internal research funding to explore this issue further.

Another area of interest is some sort of cluster-on-demand [Moore] scheme to allow use of nodes in different ways at different times. One suggestion has been to create an Emulab [White] sub-cluster which can be used for computation when not being used for network simulation.

Distributed file systems like GFS and distributed process models like BProc are an area we would like to see explored further on FreeBSD. Currently there is significant work on Linux, but little on FreeBSD.

We are working to develop new higher level parallel programming toolkits to support specific applications such as mesh generation for computational fluid dynamics models. We are currently in the process of deploying the Globus Toolkit on the Aerospace network which will potentially allow users

to run applications which span multiple computing resources including Fellowship, other Aerospace clusters, and SMP systems such SGI Origins. Such applications could be built using programming tools such as GridRPC [Seymour] being developed by the GridRPC Working Group of the Global Grid Forum.

In the mid-term we are looking toward a migration to FreeBSD 5.x for improved SMP performance and threading support. For improved threading alone, this will be an important step for us. There are some significant challenges we need to overcome, the most significant one being the need to upgrade our network-boot infrastructure to the NetBSD derived rc.d boot scripts [Mewburn] and the GEOM disk subsystem [Kamp].

Fellowship currently runs a wide mix of jobs which are used being used to make significant decisions regarding space systems. We feel that FreeBSD has served us well in providing a solid foundation for our work and is generally well supported for HPC. We encourage others to consider FreeBSD as the basis for their HPC clusters.

Acknowledgments

We would like to acknowledge the support of the GPS and STSS program offices. Additional support was provided by the Aerospace Computer Systems Division. Without their funding of administrative costs, Fellowship would not be what it is today.

References

[Becker] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer, *Beowulf: A Parallel Workstation for Scientific Computation* Proceedings, International Conference on Parallel Processing, 1995.

[Moore] Justin Moore, David Irwin, Laura Grit, Sara Sprenkle, and Jeff Chase. *Managing Mixed-Use Clusters with Cluster-on-Demand.* Department of Computer Science. Duke University.
`http://issg.cs.duke.edu/cod-arch.pdf`

[Kamp] Kamp, Poul-Henning. geom(4). GEOM - modular disk I/O request transformation framework. *FreeBSD Kernel Interfaces Manual* FreeBSD 5.1.

[Perlstein] Perlstein, Alfred. FreeBSD Jumpstart Guide.
`http://www.freebsd.org/doc/en_US.ISO8859-1/articles/pxe/`

[Mewburn] Mewburn, Luke. The Design and Implemenation of the NetBSD rc.d system.
`http://www.mewburn.net/luke/papers/rc.d.pdf`

[MPI] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.*
`http://www.mpi-forum.org/docs/mpi-11.ps`

[SGI] Silicon Graphics, Inc. *Energy Department's Blue Mountain Supercomputer Achieves Record-Breaking Run.*
`http://www.sgi.com/newsroom/press_releases/2000/may/blue_mountain.html`

[Jeong] Garrett Jeong, David Moffett. *Welcome to ACME - The Advanced Computer Matrix for Engineering.*
`http://acme.ecn.purdue.edu/`

[Schweitzer] A. Schweitzer. *The Klingon bird of Prey PC cluster.*
`http://phoenix.physast.uga.edu/klingon/`

[SciClone] The College of William and Mary. *SciClone Cluster Project.*
`http://www.compsci.wm.edu/SciClone/`

[RFC1918] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear. *Address Allocation for Private Internets.*

[RFC1178] D. Libes. *Choosing a Name for Your Computer.*

[Tolkien] J.R.R. Tolkien. *The Lord of the Rings* 1955.

[White] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. *In 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[Seymour] Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H., An Overview of GridRPC: A Remote Procedure Call API for Grid Computing. *3rd International Workshop on Grid Computing*, November, 2002.

build.sh: Cross-building NetBSD

Luke Mewburn, Matthew Green

The NetBSD Foundation

lukem@NetBSD.org, mrg@eterna.com.au

Abstract

NetBSD has a cross build infrastructure which allows cross-building of an entire NetBSD release including bootable distribution media. The build process does not require root privileges or writable source directories. The build process works on many POSIX compatible operating systems. This paper explains the changes made to NetBSD to enable this build process, enumerates benefits of the work, and introduces future work enabling cross building of any software for NetBSD.

1. Introduction

NetBSD [1] is the most portable Unix operating system in common use. It is freely available and redistributable, and runs on a broad variety of platforms from modern desktop systems and high end servers that can build an entire release in less than an hour, to embedded systems and older machines that may take several days to build a release.

In late 2001, work began on changes to improve the ability of NetBSD to be cross built, especially an entire release. This system is referred to as "*build.sh*", because that is the name of the script that is the user-visible front-end to the infrastructure.

NetBSD 1.6 was the first release to be shipped with *build.sh*, and the Release Engineering group of the NetBSD Project took advantage of it to cross-build binary releases for 39 platforms on a near daily basis during the release cycle for NetBSD 1.6 [2]. Previous releases required access to each of the various platforms by release engineers, or co-ordination with developers with that hardware. While that method works for a moderate number of platforms (NetBSD 1.5 released binaries for 20 platforms), it does not scale, especially as the number of platforms in NetBSD is growing (54 as of June 2003).

2. Background

2.1. NetBSD

Since the NetBSD project was started in 1993 it has had a goal of being portable [3] to many target platforms. There has been significant effort in designing, implementing, and improving NetBSD to make it easier to "port" to a new target platform [4]. Device drivers are written in a way that permits easy sharing between platforms without unnecessary code replication [5].

The source code portability of NetBSD did not equate to "ease of use" when building the system on a host other than the target platform, or indeed, natively.

Prior to *build.sh* a NetBSD release for a given platform was built "natively" on that platform on a version of the operating system that was "close" to the target release. There were exceptions, but these alternate processes were not simple to use nor easily automated, and had a variety of other limitations which*build.sh* addresses.

build.sh offers a level of flexibility in building NetBSD that has not been addressed by other open source operating systems.

2.2. Cross compiling Unix

Unix was cross-compiled from the beginning, but when native hosting was available, that became the main development methodology and has remained so.

Cross-compilation is the technique of running programs on a "host" system to generate object code for a different "target" system. This has not been an easy task for most system builders and has generally not been integrated into operating system build processes as used by open source operating systems.

Freely available software projects such as GCC [6] have supported being cross-compiled for a long time, and GCC is part of the GNU toolchain which NetBSD uses and is heavily dependent upon for cross-compiling.

2.2.1. An introduction to cross-compiling

There are many parts to a full cross compiler environment. Besides the compiler itself, many others tools and files are required to create functional programs. Everything that a normal compiler needs must be present. For the GNU toolchain, this includes:

- The compiler - `gcc`.
- The assembler - `as`.
- The linker - `ld`.
- The "binutils"; `size`, `nm`, `strip`, `ar`, etc.
- Header files (provided by NetBSD).
- Libraries (provided by NetBSD and the GNU toolchain).

Following is a quick overview of how it all works. This is basically the same for any compiler; in this example the details are from the GCC C compiler:

1. The C compiler front-end `gcc` calls the C pre-processor `cpp` on an input source file, usually a ".c" file, producing a ".i" file. This is still valid C code but will now be devoid of C pre-processor directives. (Actually in modern GCC, the `cpp` pass is done inside the `cc1` pass to speed up the process and provide better error reporting. This process is largely invisible to the user.)

2. `gcc` calls the back-end `cc1` with the output of `cpp` producing a ".s" file. This is a assembler source file corresponding to the input C file.

3. `gcc` calls the assembler `as` with the output of `cc1` producing a ".o" file. This is an object file corresponding to the input assembler file.

4. gcc calls the linker `ld` with the output of `as` plus several other files (sometimes collectively called the "crtstuff"), to produce an executable.

   In addition to creating executables, archive and shared libraries are built. Archive libraries are usually created with the `ar` binutils program from object files. Shared libraries are created by calling `gcc` with the `-shared` option, which calls `ld` with various options to create a shared library.

`gcc`'s `-v` flag may be used to see exactly what external programs are called. For example, cross-compiling a simple NetBSD/sparc "hello world" C program on a NetBSD/macppc box gives:

```
what-time-is-love ~> /tools/bin/sparc--netbsde
lf-gcc -I/dest/usr/include -L/dest/usr/lib -B/
dest/usr/lib/ -v -save-temps -o hello.x hello.
c
Reading specs from /tools/lib/gcc-lib/sparc--n
etbsdelf/2.95.3/specs
gcc version 2.95.3 20010315 (release) (NetBSD
nb4)
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/cp
p0 -lang-c -v -I/dest/usr/include -isystem /de
st/usr/lib/include -D__GNUC__=2 -D__GNUC_MINOR
__=95 -D__sparc__ -D__NetBSD__ -D__ELF__ -D__s
parc__ -D__NetBSD__ -D__ELF__ -Asystem(unix) -
Asystem(NetBSD) -D__GCC_NEW_VARARGS__ -Acpu(sp
arc) -Amachine(sparc) -D__sparc hello.c hello.
i
GNU CPP version 2.95.3 20010315 (release) (Net
BSD nb4) (sparc-netbsdelf)
#include "..." search starts here:
#include <...> search starts here:
 /dest/usr/include
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/in
clude
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../sparc--netbsdelf/include
End of search list.
The following default directories have been om
itted from the search path:
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../include/g++-3
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../sparc--netbsdelf/sys-include
End of omitted list.
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/cc
1 hello.i -quiet -dumpbase hello.c -version -o
 hello.s
GNU C version 2.95.3 20010315 (release) (NetBS
D nb4) (sparc-netbsdelf) compiled by GNU C ver
sion 2.95.3 20010315 (release) (NetBSD nb4).
 /tools/sparc--netbsdelf/bin/as -32 -o hello.o
 hello.s
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/co
llect2 -m elf32_sparc -dy -dc -dp -e __start -
dynamic-linker /usr/libexec/ld.elf_so -o hello
.x /dest/usr/lib/crt0.o /dest/usr/lib/crti.o /
dest/usr/lib/crtbegin.o -L/dest/usr/lib -L/des
t/usr/lib -L/tools/lib/gcc-lib/sparc--netbsdel
f/2.95.3 -L/tools/sparc--netbsdelf/lib hello.o
 -lgcc -lc -lgcc /dest/usr/lib/crtend.o /dest/
usr/lib/crtn.o
```

The output shows the `cpp0`, `cc1`, `as` and `ld` programs being called in succession. An object file is created and the sundry "crtstuff" is added to the final `ld` line.

If `gcc` is passed the `-save-temps` option the output of each program will be saved rather than deleted after it has been processed. In the above example, the following files were created:

```
what-time-is-love ~> ls -l hello.?
-rw-rw-r-- 1 mrg mrg    57 Jun 5  2001 hello.c
-rw-rw-r-- 1 mrg mrg  7820 Jul 7 15:01 hello.i
-rw-r--r-- 1 mrg mrg   872 Jul 7 15:01 hello.o
-rw-rw-r-- 1 mrg mrg   408 Jul 7 15:01 hello.s
-rwxr-xr-x 1 mrg mrg 68638 Jul 7 15:01 hello.x
```

These are the source file and the `cpp`, `cc1`, `as` and `ld` outputs, respectively.

For a cross compiler the assembler, linker and binutils must also be "cross" tools. For the GNU toolchain these are normally called "$target-$tool". For example `powerpc-eabi-ld` is the linker for the "powerpc-eabi" target.

Each cross compile environment needs to provide header files and libraries appropriate to the target platform. These may be provided either as a static set of files or the build process may produce them as part of its bootstrap. The NetBSD process uses both of these techniques for header files and builds libraries from source.

2.2.2. Targets, hosts, and build hosts

One part of cross compilers that is often confusing is the difference between a "target", a "host" and a "build host". These are:

target	The platform the toolchain creates output for.
host	The platform that runs the toolchain.
build host	The platform that builds the toolchain.

The three normal ways to build the GNU compiler are:

1. A native build is when all three host types are the same:

   ```
   ./configure && make bootstrap
   ```

2. A normal cross compile is when the "build host" is the same as the "host":

   ```
   ./configure --target=powerpc-eabi && make
   ```

3. A "Canadian cross compile" is when "build host", "host", and "target" are all different:

   ```
   ./configure --build=i386-netbsdelf \
       --host=i686-pc-cygwin \
       --target=powerpc-eabi && make
   ```

   This strange beast is called a "Canadian cross compiler", because when a name was needed there were three political parties in Canada. It is most often found when dealing with pre-built toolchains. The above example would produce a toolchain that will run on a i686-pc-cygwin host (i.e., a Windows system with Cygwin installed) and will target embedded PowerPC platforms. The toolchain would be built on a NetBSD/i386 machine.

   A normal cross-compiler is required to build a Canadian cross-compiler.

Most people who build software use a native build in all cases. Usually people using cross compilers use the same "build host" and "host".

The host compiler is the compiler used to bootstrap the cross compiler, and to build any other host tools that need to run on the build host.

3. Feature set

The features of the *build.sh* system are detailed below. Other systems have addressed some of these issues but not all of them.

3.1. Cross compilation of a NetBSD release

This is a huge benefit to developers who wish to use NetBSD on a target platform (often an embedded system) but prefer to use a different host development system.

This feature is dependent upon toolchain support as described earlier.

Suitable build systems include:

- Current and previous NetBSD releases.
- Other Unix-like systems, such as Darwin (Mac OS X), FreeBSD, HP-UX, Linux, and Solaris.
- Windows with Cygwin.

3.2. Simplicity

build.sh needs to be simple to use while retaining enough flexibility to allow advanced users and developers to customize their build environment.

NetBSD has a finite amount of volunteer resources, so simplifying the build process reduces the support load.

3.3. Read-only source trees

The source tree can be read-only and can be shared between builds for different target NetBSD platforms. This permits building directly from read-only media or "mirrors" of source code without causing unnecessary mirroring conflicts. In this case the build output is written to a separate writable section of the file system.

There are no restrictions besides path-name length on the file system location of the source tree or the object tree.

3.4. Root privileges are not necessary

No root or other special privileges are required in order to build distribution media. Previous build processes required privileges to create tar files or file systems that contained devices, files owned by other users, set-ID files, etc.

Most other build systems require special privileges. For example, the Debian fakeroot [7] mechanism requires the host operating system to support shared object libraries with $LD_PRELOAD functionality (where a shared object is loaded after the main program but before the program's shared object dependencies), and this breaks the next requirement.

3.5. Avoid non-portable OS features

Prior to *build.sh*, the creation of NetBSD boot media required root privileges to create a "loopback vnode disk device" (`vnd`), install boot blocks, create a file system on the vnd, mount the file system, and create file system entries owned by the non-building user. Most other systems still build their distribution media in this manner.

build.sh does not require host operating system features such as: virtual disk devices, loopback file systems, chroot cages, shared objects/libraries, or dynamically loaded modules.

3.6. Segregated build tools

build.sh builds "host" tools to $TOOLDIR, which is a location separate to the host system's native tools. *build.sh* uses the tools in $TOOLDIR to build the NetBSD system installing to a separate $DESTDIR location. This removes the "chicken & egg" problem that can occur when the host platform's in-tree tools would need to be upgraded to build a newer source tree.

The older technique using the in-tree toolchain had the drawbacks:

- Without massive contortions, the build host had to be running NetBSD of the same architecture and similar software vintage.
- The in-tree toolchain required periodic updating.
- Updating the in-tree toolchain often required special handling that could not be easily automated. For example, at one time */usr/share/mk* needed `make`, and at another time `make` needed */usr/share/mk* to be updated.

3.7. Minimize impact on NetBSD source

There was minimal impact on the existing NetBSD sources and build infrastructure.

We did not want to change the NetBSD source tree to support being built by foreign compilers. Accordingly the host's native toolchain is only used to build the NetBSD host tools.

It was necessary to improve the portability of the NetBSD host tools to non-NetBSD platforms. These changes were not significant.

3.8. Don't special-case cross-compiles

It was a requirement that *build.sh* was usable for native builds as well as for cross compiling.

This feature is another component of the effort to reduce the support overhead of building NetBSD, whilst retaining our desired flexibility.

4. Results

4.1. Benefits

4.1.1. Regular automated builds

During the NetBSD 1.6 release engineering process, 39 platforms had near daily builds of the base operating system release, ensuring that at least all build problems were well known every day, and that up to date builds were available for testing by anyone. The majority (37) of these builds were performed on a dual processor AMD MP2000+ machine generously donated by AMD and Wasabi Systems, and the alpha and sparc64 platforms were built on Matt Thomas'CS20 alpha.

This contrasts with the NetBSD 1.5 release where 20 platforms had a binary release; all of them were built natively on the particularly port, requiring each port to have a dedicated person to build it, with 12 days time from when the first port was finished and available to the last port.

4.1.2. Simplicity of use

The new *build.sh* is very simple and easy to use, yet powerful. Anybody can build a kernel, userland or full release for most platforms on the fastest machine they have available.

4.1.3. More portable source code

It has identified every tool in NetBSD used for building. We know what tools we need and what features these tools must support. We had to make sure these tools were portable programs that would build and run correctly on other platforms.

4.1.4. Cross-platform builds

We no longer require running a particular version of NetBSD in order to be able to build it. Building NetBSD 1.6 from a NetBSD 1.5 machine is now supported, as is building from Solaris, Linux and more. Gone are the days of source build bootstrap problems that has plagued NetBSD since day one.

4.2. Costs

4.2.1. Teething problems

build.sh took a long time to settle down and be stable and useful. On the other hand, adding new platform support is no deal now, this is a one time cost that has been paid.

4.2.2. Cross-unfriendly software

Much software is written without consideration to cross compiling and thus makes it very difficult to cross compile. Not only the way the software itself is written, but the way the software is actually built. Many software build processes build helper programs to generate real code, and so these programs must not only work correctly for the target system rather than the native system, they must be built by the "host" compiler. The automounter `amd` was affected by this, as the build process assumed the local host architecture was the same as the program being built, causing a sparc binary built on i386 to incorrectly byte swap data into little endian for processing.

4.2.3. CPU endianness and word size issues

Endianness and CPU word size issues within the toolchain can cause problems. Due to bugs in the version of GCC that we use (2.95.3), NetBSD/alpha can not be cross compiled from a 32-bit host, and NetBSD/i386 can not be cross compiled from a 64-bit host.

Fortunately, we know that with GCC 3.3, NetBSD/alpha can be built on 32-bit platforms, and NetBSD's version of GCC will be upgraded to 3.3.x before NetBSD 2.0 is branched.

4.2.4. Overhead in converting a platform

Not every platform has been converted to *build.sh* yet. In some cases (e.g., pc532, playstation2) the in-tree toolchain does not support those platforms in the *build.sh* framework, whilst in others the platforms may not have had a complete build infrastructure before *build.sh* was integrated, and there were limited resources prior to the NetBSD 1.6 release.

4.2.5. Performance implications

There is a time cost in compiling the host tools, but it is not considered significant enough to pose a problem. For example, on an AMD XP2500+, the times for the various methods of building an i386 release are:

Time to build tools with *build.sh*: `./build.sh -U tools`	5m 6s
Time to build a release with *build.sh*: `./build.sh -U release`	1h 28m
Time unsuccessfully spent attempting to `make release` with the old build process on a two month old system, which failed due to problems with out-of-date tools.	> 2h

5. Implementation

The implementation of the *build.sh* infrastructure comprises of a variety of elements.

The build process uses various environment variables to control its operation, including:

TOOLDIR	Pathname to host tools for a given host platform.
DESTDIR	Pathname to built NetBSD system.
RELEASEDIR	Pathname to resulting NetBSD release files.

5.1. src/build.sh

build.sh is a Bourne shell script designed to build the entire NetBSD system on any host with a POSIX compliant Bourne shell in */bin/sh*. It creates a directory for various host tools uses to cross build the system (referred to as $TOOLDIR), creates a "wrapper" to `make` to pass in various settings, builds the host tools with the make wrapper, and uses the host tools to build the rest of the system, into a staging area referred to as $DESTDIR.

build.sh is very flexible and can be used to build an entire release, a specific kernel, just the tools or make wrapper, or even upgrade the installed system from a previously populated $DESTDIR.

The order of operation of *build.sh* is:

1. Validate arguments including the target platform (which defaults to the host platform) which is stored in $MACHINE, and the target machine (CPU) architecture which is stored in $MACHINE_ARCH.

2. Build a host binary of `make` and install as *$TOOLDIR/bin/nbmake*.

3. Create the "make wrapper" shell script that contains various (environment variable) settings which control the build and install as *$TOOLDIR/bin/nbmake-$MACHINE*. (This may be the last operation performed for this invocation of *build.sh*.)

4. Build the host tools to generate the binaries for the target $MACHINE and install under $TOOLDIR, with the executable host binaries available in *$TOOLDIR/bin*.

5. Perform any other operations requested using the $TOOLDIR host tools, including:

   - Build a NetBSD distribution into $DESTDIR.
   - Build all the kernels for a specific NetBSD release and package those and the contents of the $DESTDIR into a release under $RELEASEDIR.
   - Build a specific kernel.
   - Create the release "sets" from $DESTDIR or the source "sets" from the source tree.
   - Upgrade a directory (usually "/") from $DESTDIR.

Certain $MACHINE platforms support more than one $MACHINE_ARCH CPU architectures (a bi-endian processor or are processor with more than one word size are considered to be separate CPU machine architectures), and *build.sh* supports building the different target CPU architectures as separate releases.

5.2. src/BUILDING

In-tree documentation for *build.sh*.

5.3. src/Makefile

This Makefile at the top-level of the source tree contains various targets to facilitate building the entire NetBSD source tree, including:

build	Build the entire NetBSD system, in an order that ensures that prerequisites are built in the correct order.
distribution, buildworld	Perform `make build` and then install a full distribution into $DESTDIR, including the contents of *$DESTDIR/etc* and *$DESTDIR/var*.
release	Perform `make distribution`, then builds kernels, distribution media, install "sets", and then packages the system under $RELEASEDIR.
installworld	Install the distribution from $DESTDIR to $INSTALLWORLDDIR (which defaults to "/").

5.4. src/share/mk

To simplify the build process, NetBSD uses a library of `make` Makefile include files (with the suffix ".mk") in *src/share/mk*. This was inherited from the 4.3BSD Networking/2 and 4.4BSD Lite releases, and has been significantly enhanced in the decade since.

5.5. src/tools

Traditionally, BSD systems are built using in-tree tools, compilers, include files and libraries. This isn't usable for cross-compilation, and has many other problems which this new infrastructure fixes, as documented in section 3.

src/tools contains the `make` infrastructure required to build various host tools used during the build. These tools are built using an autoconf-built compatibility frame-work, and use "reach over" Makefiles into the rest of the NetBSD source tree to minimize replication of code. The source code for these host tools have been slightly modified to allow them to be built in as a normal NetBSD (target) program, as well as a host tool.

The host tools are installed into $TOOLDIR, and the BSD `make` ".include" infrastructure in *src/share/mk* selects the various host tools in preference to the "standard" versions. For example, for an i386 target $CC will be set to *$TOOLDIR/bin/i386--netbsdelf-gcc* instead of *cc*.

There is support for using an external toolchain rather than the in-tree versions of binutils and `gcc`. This is useful when there isn't yet support in the in-tree toolchain for a target platform.

The host tools currently consist of:

```
as asn1_compile binutils cap_mkdb cat cksum
compile_et config crunchgen ctags db dbsym
file gcc gencat groff hexdump install
installboot ld lex lint lorder m4 makefs
makewhatis mdsetimage menuc mkcsmapper
mkdep mkdep mkesdb mklocale mktemp msgc
mtree pax pwd_mkdb rpcgen sunlabel texinfo
tsort uudecode yacc zic
```

The majority of the host tools are installed with an "nb" prefix, to differentiate them from similarly named commands on the host. The exceptions are the GNU toolchain programs, which already have a name such as `i386--netbsdelf-gcc`.

5.6. METALOG support

Traditionally, `install` is run as root to install paths into the appropriate location under $DESTDIR (which defaulted to "/"), with the appropriate ownership and permissions.

This prevents non-root users from building a full distribution, as many files need specific permissions, such as setting set-user-ID root on */usr/bin/su*.

To solve this issue, we enhanced the specification file for the existing `mtree` tool to support a full path name (versus a context-sensitive relative path name), and referred to the result as a "metalog" entry. An example entry is:

```
./usr/bin/su type=file mode=04555 \
    uname=root gname=wheel \
    time=1057493599.102665
```

For a given build, a *METALOG* file is created, with a "metalog" line for each installed path name. The *METALOG* file is manipulated and parsed by various tools as necessary throughout the full build process.

`install` was modified to optionally install the paths as the current user and without any special permissions, and instead log the requested permissions to the *METALOG*. The ".mk" files in *src/share/mk* and a small number of special case Makefiles were all that needed to be modified to take advantage of this support in `install`.

`pax` was modified to support parsing a *METALOG* for the list of paths to add to an archive, and even add "fake" entries for devices which may not have been created in $DESTDIR. This is used to build *tar.gz* files which contain the correct ownership and permissions from a *METALOG* and $DESTDIR populated by a build by an unprivileged user. The scripts that create the "installation sets" were enhanced to parse the *METALOG* and invoke `pax` appropriately.

5.7. makefs

Various platforms use distribution media which require a ffs file system to boot from. Previously, NetBSD built these using a "loopback vnode disk driver" (`vnd`), which is not available on many other systems, and requires root privileges to mount and write to in any case.

`makefs` was added to create a file system image from a directory tree and optionally a *METALOG*. It is conceptually similar to `mkisofs`, a GPL-ed application which creates ISO-9660 file system images.

With `makefs`, an unprivileged user can create a ffs file system, complete with device nodes (with the latter being specified in a *METALOG*).

While `makefs` has been written in a manner that easily supports the addition of different file system types, it currently only supports creating ffs file systems, in either little or big endian (since NetBSD's ffs code supports opposite endian ffs file systems, c.f. the "FFS_EI" kernel option, and support in userland tools such as `newfs` and `fsck_ffs`). Support for other file systems such as iso9660, ext2fs, and FAT has been considered, but is not a high priority at this time.

The implementation of the ffs back-end re-uses a reasonable amount of the existing source code in the current NetBSD kernel implementation of ffs (in *src/sys/ufs/ffs*), but for simplicity, functions such as the block allocation code were re-implemented in a simpler manner. Various assumptions were made as part of this process, including that block reallocation would not be necessary, since the size of all files and directories (as files) is known at file system creation time.

While ffs was originally implemented as a user process before its integration into the 4.2BSD kernel, after two decades of kernel hacking it is heavily tied to the buffer cache and other kernel sub-systems, which makes it more difficult to use in a stand-alone program.

5.8. installboot

Most platforms need boot blocks on their distribution media, and these are installed with `installboot`. Previously, each platform had its own version of `installboot` in */usr/mdec/installboot*, which was compiled as part of, and heavily dependent upon, the kernel sources for that platform. They also required root privileges and generally required kernel support for specific disk-label `ioctl()`'s and only worked on actual disk devices.

/usr/sbin/installboot is a replacement for the machine-dependent versions of `installboot`. It can function on file system images, and doesn't need root privileges or kernel support for specific `ioctl()`'s to do so. The design of the boot blocks between the various platforms has been standardized as well, even if the actual implementation is different due to obvious differences in platform hardware.

All platforms that shipped with binary releases for NetBSD 1.6 (except i386) were converted to this new infrastructure. i386 was a special case due to the baroqueness of the implementation of `installboot` for that platform, but as the "cross build" host for the i386 release was a dual processor i386 box, it could run the tool "natively". This has been resolved in NetBSD-current.

On a related note, it is possible to make CD-ROMs that boot NetBSD on multiple platforms. For example, i386, sparc64, and macppc can boot off the same disk, and there's ample room for other platforms. NetBSD 1.6, with the 39 platforms that shipped with a binary release and associated source, fit on 4 CD-ROMs and booted on 9 of them.

5.9. src/etc/postinstall

`postinstall` is a script that checks for and/or fixes configuration changes that have occurred as NetBSD has evolved.

`postinstall` was added to the build system to detect, and in most cases, automatically fix, changes to configuration that must be performed due to software changes.

The tests that `postinstall` supports are:

postinstall	*/etc/postinstall* is up to date
defaults	*/etc/defaults* is up to date
mtree	*/etc/mtree* is up to date
gid	*/etc/group* contains required groups
uid	*/etc/passwd* contains required users
periodic	*/etc/{daily,weekly,monthly,security}* is up to date
rc	*/etc/rc** and */etc/rc.d* is up to date
ssh	`ssh` and `sshd` configuration update
wscons	`wscons` configuration file update
makedev	*/dev/MAKEDEV* is up to date
postfix	*/etc/postfix* is up to date
obsolete	obsolete file sets
sendmail	`sendmail` configuration is up to date

6. Future Work

6.1. xsrc

NetBSD does not currently support a cross-build of "xsrc" (our copy of X11R6 / XFree86 4.x), but this will be fixed eventually. XFree86 4.3 added its own support for cross-compiling. We may implement our own method of cross compiling X11 for better integration with our build system. XFree86's cross-build solution does not address all of our requirements, especially the removal of the need for root privileges.

6.2. pkgsrc

"pkgsrc" (the NetBSD packages collection) is not cross buildable. A smaller number of particular, probably smaller packages would probably fairly easy to get cross buildable, but the vast majority would each require significant effort.

There have been two suggestions to simplifying the solution to this problem:

1. Krister Walfridsson has proposed building the packages natively in an emulator, using optimizations such as only emulating user-mode, and capturing system calls and running those natively (after manipulating the arguments).

   An enhancement to that is to detect if an emulated program is `gcc` (for example), and invoking the host's cross-compiler natively for that case.

   The initial progress looks very promising, with an arm emulator compiling six times faster than the native platform.

   This proposal would be acceptable for solving the "xsrc" problem in section 6.1.

2. Use a tool such as distcc [8] to distribute the compilation of C or C++ code to (faster) remote systems, even if those systems are different to the current system, as long as an appropriate cross-compiler is available. This would still require the use of the native system for part of the build process.

7. Conclusion

build.sh has been an extremely useful solution to a variety of problems. It's now much simpler to build NetBSD releases from systems running earlier NetBSD releases, and even build on other systems such as Darwin / MacOS X, FreeBSD, Linux, and Solaris.

Generally, the support issues in NetBSD using *build.sh* have been less than for previous releases, especially considering the number of platforms now supported. Previously, it was extremely important to upgrade various in-tree tools, includes, and libraries in a specific order (that often changed) before completing the rest of the build. Now, a full build can be made without impacting the running system, and then an upgrade easily performed once a successful build is available.

The authors regularly use *build.sh* to cross-build entire releases on our (faster) alpha, i386, macppc, and sparc64 systems for platforms such as alpha, i386, macppc, pmax, shark, sparc, sparc64, and vax, as an unprivileged user using read-only source.

Acknowledgements

Todd Vierling wrote the original version of*build.sh*, and setup the *src/tools* framework.

Jason Thorpe does a **lot** of work in the toolchain and associated *build.sh* infrastructure.

Erik Berls wrote and maintains the autobuild script that is used on the NetBSD Release Engineering machines to automatically build multiple release branches for multiple target platforms on multiple build machines.

Wasabi Systems, Inc. funded a lot of the development of this work by paying the salaries of various developers.

FSF provides the GNU toolchain which is so nicely portable and cross compile friendly.

References

[1] *Welcome to the NetBSD Project*,
http://www.NetBSD.org/

[2] *NetBSD Project Autobuild [for NetBSD 1.6]*,
http://releng.NetBSD.org/ab/B_netbsd-1-6/

[3] *Portability and supported hardware platforms*,
http://www.NetBSD.org/Goals/portability.html

[4] Frank van der Linden, *Porting NetBSD to the AMD x86-64: a case study in OS portability*, BSDCon 2002,
http://www.usenix.org/events/bsdcon02/full_papers/linden/linden_html/

[5] Jason R. Thorpe, *A Machine-Independent DMA Framework for NetBSD*, Usenix 1998 Annual Technical Conference,
http://www.usenix.org/publications/library/proceedings/usenix98/freenix/thorpe.dma.ps

[6] *Welcome to the GCC home page*,
http://gcc.gnu.org/

[7] *Package: fakeroot 0.4.4-9.2*
http://packages.debian.org/stable/utils/fakeroot.html

[8] *distcc: a fast, free distributed C/C++ compiler*,
http://distcc.samba.org/

GBDE - GEOM Based Disk Encryption

Poul-Henning Kamp
The FreeBSD Project
phk@FreeBSD.org

Abstract

The ever increasing mobility of computers has made protection of data on digital storage media an important requirement in a number of applications and situations. GBDE is a strong cryptographic facility for denying unauthorised access to data stored on a "cold" disk for decades and longer. GBDE operates on the disk(-partition) level allowing any type of file system or database to be protected. A significant focus has been put on the practical aspects in order to make it possible to deploy GBDE in the real world. [1]

1. Losing data left and right

In the last couple of years, gentlemen of the press have repeatedly been able to expose how laptop computers containing highly sensitive or very valuable information have been lost to carelessness, theft and in some cases espionage. [THEREG]

The scope of the problem is very hard to gauge, since it is not a subject which the involved persons and, in particular, institutions are at all keen on having exposed. However, a few data points have been uncovered, revealing that the U.S. Federal Bureau of Investigation loses, on average, one laptop every three days. [DOJ0227]

When a computer is lost, stolen or misplaced, it is very often the case that the computer hardware represents a value which is insignificant compared to the value of the disk contents. More often than not, the only reason the press heard about it was that the material on the disk was "hot" enough to make the loss of control rattle people at government level.

While it is easy to blame these incidents on "user error", as is generally done, doing so makes it a very hard problem to fix. Human nature being what it is, seems to remain just that.

In the absence of technical counter measures, administrative measures have been applied, generally with abysmal results. In one case, a bureaucracy has handled the problem according to what could easily be mistaken for the plot from a classic Buster Keaton movie:

> First a laptop was forgotten and lost in a taxi-cab. New policy: always drive your own car if you bring your laptop. Then a car was stolen, including the laptop in the trunk. New policy: always bring your laptop with you. The next laptop was stolen from a pub while the owner was bowing to the pressures of nature. New policy: employees are not to carry their own laptops outside the office at any time. Laptops will be transported from and to the employees home address by the agency security force and will be chained and locked to a ring in the wall installed by the company janitors. All requests must be filed 3 days in advance on form ##-#. [PRIV]

2. Protecting disk contents

Protecting the contents of a computer's disk can in practice be done in two ways: by physically securing the disk or by encrypting its contents.

Physical protection is increasingly impossible to implement. It used to be that disk drives could only be moved by forklift, but these days a gigabyte disk is the size, but not quite yet the thickness, of a postage stamp. While computers can be tied down with wires and bars can be put in front of windows, such measures are generally not acceptable, or at least not judged economically justified in any but the most sensitive operations.

That leaves encryption of the disk contents as the only practical and viable mode of protection, and both the practicality and the viability has been somewhat in doubt.

Until recently, nearly all aspects of cryptography were a highly political issue, this has eased a lot in the last couple of years and there now "only" remain a number

[1] This software was developed for the FreeBSD Project by Poul-Henning Kamp and NAI Labs, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

of rather fundamental questions in the area of law enforcement and human rights, which are still unsettled.

With the political issues mostly out of the way, the next roadblock is practical: While use of cryptography can never be entirely transparent, the overhead and workload it brings must be reasonable.

2.1. Application level encryption

Encryption at the application level has been available for a number of years, primarily in the form of the PGP [PGP] program. This is about as intrusive and demanding as things can get: the user is explicitly responsible for doing both encryption and decryption and must enter the pass-phrase for every operation. [2]

Apart from the inconvenience of this extra workload, many organisations would trust their users neither to get this right nor even to want to get it right. From an institutional point of view it is important that cryptographic data protection can be made mandatory.

2.2. Filesystem level encryption

Encryption at the file system level is a tried and acknowledged method of providing protection, but it suffers from a number of drawbacks, mainly because no mainstream file systems offer encryption.

Encrypting file systems are speciality items, which means increased cost and system administration problems of all sorts.

And since practically all operating systems use their own file system format, cross platform fully functional file systems are very rare. This means that a typical organisation will have to operate with a handful of different methods of encryption, which translates to system administration overhead, user confusion and extra effort to pass security and ISO9000 audits.

A secondary, but increasingly important issue is that data which are stored in databases on raw disk, operating system paging areas and other such data are not protected by a cryptographic file system. To protect these would mean adding yet another set of encryption methods, which leads to a situation which is very hard to handle practically and administratively.

Finally, file systems have a complex programming interface to the operating system, which traditionally has been subject to both version skew and compatibility problems.

[2] Interestingly, this is so impractical in real world use that various applications with PGP support resort to caching the pass-phrase at the application level, thereby weakening the protection a fair bit.

2.3. Disk level encryption

Encryption at the disk level can protect all data, no matter how they are stored, file system, database or otherwise.

To a user, encryption at the disk level would require authentication before the computer can be used, everything functioning transparently thereafter, with all disk content automatically protected.

Given that the programming interface for a disk device is very simple and practically identical between operating systems, there are no technical reasons why the same implementation could not be used across several operating systems.

All in all this is a close to ideal solution from an operational point of view.

There are significant implementation issues however. In difference from the higher levels, encryption at the disk level has no way of knowing a priori which sectors contain data and which sectors do not; neither is knowledge available about access patterns or relationships between individual sectors.

Where application level or file system based encryption schemes can key each file individually, a disk based encryption must key each and every sector individually, even if it is not currently used to hold data.

It has been argued that the encryption ideally should happen in the disk-drive, and while there are steps in this direction, they do unfortunately seem to have been made for the wrong reasons by the wrong people [CPRM], and have consequently not gained acceptance.

Provided the owner of the computer remains in control of the encryption, I see no reason why encryption in the disk drives should not gain acceptance in the future.

3. Why this is not quite simple

Several implementations have been produced which implement a disk encryption feature by running the user provided passphrase through a good quality one-way hash function and used the output as a key to encrypt all the sectors using a standard block cipher in CBC mode. A per sector IV for the encryption is typically derived from the passphrase and sector address using a one-way hash function. Two typical examples are [CGD] and [LOOPAES].

Unfortunately this approach suffers from a number of significant drawbacks, both in terms of cryptographic

strength and deployability.

For data to stay protected for decades or even lifetimes, sufficient margin must exist not only for technological advances in brute force technology, but also for theoretical advances in cryptoanalytical attacks on the algorithms used.

Protecting a modern disk, typically having a few hundred millions of sectors, with the same single 128 or 256 bits of key material offers an incredibly large amount of data for statistical, differential or probabilistic attacks in the future.

Worse, because the sectors contain file system or database data and meta data which are optimised for speed, the plaintext sector data typically have both a high degree of structure and a high predictability, offering ample opportunities for statistical and known plaintext attacks.

This author would certainly not trust data so protected to be kept secret for more than maybe five or ten years against a determined attacker.

But far more damning to this method is that there can only be one single passphrase for the disk. This effectively rules out the ability for an organisation to implement any kind of per-user or multilevel key management scheme: the only possible scheme is "one key per disk".

Add to this that to change the passphrase the entire disk would have to be decrypted and re-encrypted, and we have a model which may work in theory, and can be made to work in practice for a determined individual, but which would fast become an operational liability for any organisation.

4. Designing GBDE

The initial design phase of GBDE focused on determining a set of features which would make it both possible and practical for people and organisations to deploy GBDE in routine use.

The first decision has already been described and argued for: Encryption must happen at the sector or "raw disk" level, in order to be comprehensive, universal and portable.

The second decision was dictated by the fact that all security policies we have ever seen, contain a rule which says "passwords must be changed every N {days,weeks,months}". This is sound thinking, and GBDE should support it. While changing the passphrase does not necessarily have to be instantaneous, decrypting and re-encrypting an entire disk would likely take more than a day with currently available hardware, and is consequently out of the question.

The third design criterion came from the fact that people forget passphrases, and while loosing the entire content of the disk as punishment could be seen as a large-calibered educational device, it is not acceptable from a real world perspective: there must be some kind of multiple access paths.

Given that GBDE is open source software, there is little more than symbolic value in a hierarchical set of passphrases: changing the source code to bypass the hierarchy cannot trivially be prevented. It is also not clear what the hierarchy would control in the first place. Since all sectors are treated the same, it cannot be used to implement hierarchical access levels, and implementing a hierarchy which only affects the key hierarchy would be silly. It was therefore decided that GBDE would support a number of parallel key paths, and four was chosen as the default number which can be changed at compile time.

The fifth usability criterion was that GBDE should be able to use any byte string as passphrase, and not be limited to NUL terminated text strings. In many settings, the necessary key strength cannot be derived from a keyboard entered text string, and therefore it should be possible to use storage devices like USB-keys or smart cards to contain or contribute to the passphrase.

The final usability criterion was that GBDE should offer the best possible protection also for the user. This might require some explanation:

4.1. Protecting the user

The weakest link in any cryptographic deployment is almost always the users and their handling of key material.

As an example of this, most banks use two-man procedures to protect their vaults. Not because this prevents their employees from embezzling money — there are plenty of ways they can do that — but because it protects the employee from being a weak link which could be broken by means of physical violence, blackmail or hostage taking.

These kinds of situations mandate that analysis treat the user as a component, something which have become routine by now.

Unfortunately it is still a relatively common mistake to either leave the attacker out of the analysis, or to assume a very weak or even stupid attacker.

An example of this, is the "The Steganographic File System" [STEGFS], which provides a facility where a hierarchy of passphrases protect data at different levels.

The argumentation more or less goes "protect a couple of unimportant files at the lowest level, and your important files at higher levels. When captured, give them the lowest level key and deny that any more keys exist."

If we include the attacker in the analysis, she will soon know that the facility used is STEGFS, and consequently that multiple levels of keys are not only a possibility but to be expected: Why else would people use STEGFS in the first place ?

A user caught with a STEGFS encrypted set of data, will therefore likely be subject to pressure to release keys until the attacker is satisfied that there are no more keys. If the attacker is the police, this can now land the user in jail for up to five years in certain countries.

But even worse: if the attacker has her own ideas of what is to be found in the protected data, for instance information on weapons of mass destruction, but no such information is on the protected set, there is no way the user can "get off the hook" and prove his innocence: STEGFS provides no way of proving the nonexistence of any further keys.

If one studies the evidence of a real-world scenario: the plight of the junior CIA operative resident and later hostage at the Tehran embassy [IRAN], a couple of interesting insights emerge.

Very often, the user will have a tiny window in time during which it may be possible to manually activate data destruction mechanisms, or alternatively, be able to out-wait a specific timeout after which automatic mechanisms will have destroyed the data if they are not rearmed correctly.

For such a feature to offer the user effective protection it must provide a tangible feedback to the attacker that the user has destroyed the data, and can not bring it back. Effective feedback has been smoking piles of ash, but not, as related in the narrative from Tehran, piles of shredded documents.

The final GBDE design criterion was therefore that GBDE should be able to rapidly destroy key material in such a way that it can be proven that there is no hope to recover the data short of very expensive brute force methods.

5. What GBDE does not do

With any cryptographic facility, it is as important to know what it does as what it does not, this section will describe a couple of major issues which GBDE leaves to the user to solve.

5.1. Key management

It has been said that there is only one really hard problem in cryptography: the problem of key management. GBDE does not try to address that.

In all of the following we will assume that the passphrases and other key material have been handled in a sensible manner, fitting the importance of the data they are used to protect. If you put a yellow sticky note with the passphrase on a disk, no amount of cryptographic code will offer any protection.

5.2. No protection for "hot" disks

GBDE is only designed to protect data on a "cold disk". A cold disk is defined as a disk disassociated from the keys which provide access to the disk. [3]

If the GBDE protected device has been "attached" on a running system, parts of the key material will be present in memory and therefore GBDE's cryptographic protection is no stronger than the protection the operating system gives to those data.

If the protected device is not "attached", there is no key-material present in RAM and the disk is fully protected by GBDE.

A disk on a shelf or in a courier bag is obviously also a cold disk.

It is important to note that disks in laptop computers which are only "suspended" can not be be considered cold if they are attached in the kernel, because the key material is present in RAM or possibly in the "suspend to disk" partition on the disk. [4]

While most computers offer facilities for password protection when leaving suspend mode, it should be noted that modern hardware facilities may contain wide open back doors around that protection.

For instance, FireWire/IEEE1394 capable devices are mandated to have have built in "OHCI" facilities which allow direct access to the entire memory space without the CPU being involved or aware of this. We

[3] Under carefully controlled circumstances GBDE can also be used as a component to build protection for "hot disks"; that is discusse later.

[4] Interestingly, if GBDE was implemented in the disk drive and the BIOS handled the pass-phrase entry, the suspend-to-disk partition would also be protected.

estimate that using this method it would be possible to undetectably grab a snapshot of all RAM on a typical notebook computer in a few minutes. A great tool for debugging, a nightmare tool for security.

6. Cryptographic design

The usability requirements produces the overall layout of the GBDE cryptographic design which the cryptographic design must respect: A passphrase gives access to a copy of the master-key material, which again gives access to the protected sector contents.

From a cryptographic point of view, the requirements are very simple: make it as strong as possible using as few resources as possible.

While we currently have great faith in algorithms like the Rijndael/AES, and it is generally accepted that 128bit symmetric keys offer practically impenetrable security, the history of the cryptography has shown that significant weaknesses take long time to uncover.

Since the aim is that GBDE be able to protect a cold disk for a decade or more, it would be unwise to rely on our current understanding of the subject and the algorithms to remain unchallenged for the lifetime of the data.

A number of defensive measures have therefore been been included in the design criteria, in order to put sufficient margin into the cryptographic strength of GBDE.

The first criterion is that plaintext sector data should be encrypted with one-time-use (pseudo-)random keys. This measure, while relatively expensive, effectively stops differential plain-text attacks.

This forces yet a design decision since the PRN sector-keys obviously must be encrypted and stored on the disk with the encrypted plain-text. The second criterion is that these sector-keys are encrypted with a per sector "key-key" derived from only a fraction of a very large master key.

By only allowing a fraction of the master-key into the calculation, a penetration which manages to find the key-key for a given sector will not expose the master key, and since the sector key which the key-key encrypts is (pseudo-)random, no differential leverage exists at this level either.

The third cryptographic criterion was that a remapping of sectors should happen, so that the location of a particular plain-text sector on the encrypted volume is not predictable. Many file systems have well defined and easily predictable "super blocks" and allocation bitmaps which have well defined sector addresses. By mapping the locations on the disk, these sectors offer no leverage for attacks, until their encrypted location has been found by some other means.

The final cryptographic criterion was to use only standard cryptographic algorithms in good standing: Rijndael/AES as block cipher, RSA2/512 as strong one way hash and MD5 as "bit blender".

7. The GBDE algorithm

As can be seen from the above, the *a priori* requirements have closely circumscribed the solution space for GBDE, and the algorithms four steps follows more or less directly from the design.

7.1. From passphrase to master key

It is generally accepted that a passphrase consisting of one or more sentences in a human language only contain about one bit of effective entropy per character, and obviously they are variable length. Therefore the passphrase is passed through the SHA2/512 one-way hash algorithm. This transforms the variable length passphrase to 512 bits which contain as much as possible of the entropy in the passphrase.

These 512 bits, called "the key material", are used to locate and decrypt the so called "lock sector" which contains the master-key and various parameters.

A compile time constant, sets the number of lock sector copies supported, the default number is four. These copies are located in four sectors chosen randomly at the time when the device is initialised for GBDE usage.

The first 128 bits of the key material are used to decrypt the sector offset of the encrypted and encoded lock sector. The encrypted version of this offset can either be stored in the first sector of the device or outside the device in a file.

Once read from disk, the next 256 bits of the key material is used to decrypt the encoded lock sector using AES/CBC/256.

The final 128 bits of the key material define the order in which the fields of the lock sector are encoded. Numeric fields are encoded in little-endian byte order so that the on-disk format is portable between different architectures. Once decoded, one of the fields offer a MD5 hash checksum which can be used to prove that this is actually a valid lock sector.

7.2. Sector mapping

It follows from the above that a number of transformations are necessary on the sector location, in

addition to the cryptographic transformation of the sector contents.

The first transformation, splits the plain-text sectors into a number of zones. The size of a zone is chosen so that one sector exactly contains the encrypted sector keys for all the payload sectors in the zone. With a sector size of 512 bytes and a sector key size of 128 bits, the results in a zone size of

$$\frac{512bytes/sector * 8bits/byte}{128bits} = 32sectors$$

The extra sector with the encrypted sector keys is appended after the last of the 32 data sectors, making the zone 33 sectors long in the encrypted image.

It should be noted that the sector size for the encrypted device and consequently the zone size is a parameter set at device initialisation time.

The second transformation is a rotation by an integral number of sectors. The number is randomly chosen at device initialisation time and recorded in a field in the lock sector. This step addresses the design requirement that the location of the encrypted sector should not be trivial to determine.

The third transformation inserts the four lock sectors into their locations. The locations of these are chosen at random at device initialisation time, and stored in fields in the lock sector. It follows quite obviously from this, that each lock sector must contain information about the location of all lock sectors, in order to map around them.

Finally an offset is added which determines where the encrypted image starts in the underlying device.

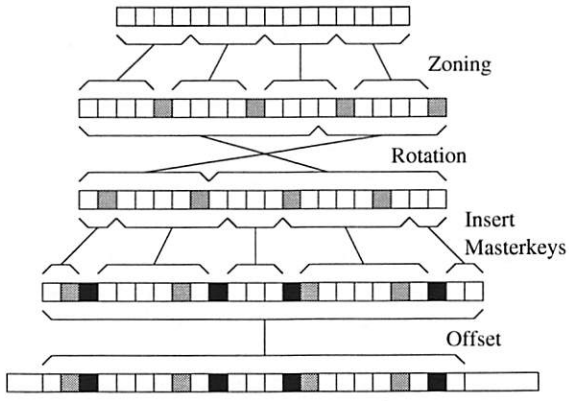

fig 1: mapping operations.

The offset mapping serves two purposes, it allows the first sector of the underlying device to be reserved to contain the encrypted locations of the lock sectors, but it also allows the administrator to use only a part of the underlying device for GBDE encrypted data.

7.3. Plausible denial

This could be used to implement "plausible denial" by embedding the GBDE partition inside some data which credibly can be claimed to be something else. Since high entropy data are rare in the wild, special care is needed to build a convincing cover story to explain the existence of otherwise unexplained random looking bits.

Most computers today come pre taxed with a Microsoft operating system, and this could be used for the cover story: Flush all free space in the Windows partition with random bits, locate a long stretch of free space in the partition and use that to contain the GBDE encrypted data. If no other evidence betrays the existence of the GBDE partition, it should be possible to deny its existence.

A truly paranoid setup would leave the computer configured to boot the Windows system by default, and locate the GBDE data in such a way that it would be destroyed by the act of doing so.

7.4. "Key-key" derivation

For each sector to be processed, we need to derive the key-key which the sector key is encrypted with. For this purpose the lock sector contains two fields, the "salt" which is a 128 bit (pseudo-)random number, and the master key which is a 2048 bit (pseudo-)random number. Both of these are generated when the device is initialised for GBDE usage and not subsequently changed.

In order to make the format architecture invariant, the the sector address is encoded as a little-endian 64 bit integer

$$sect = LE64(sectoraddress)$$

and then, surrounded by the salt, run through the MD5 algorithm which acts as a "bit-blender". The salt is necessary to prevent pre-computation of a dictionary of all possibly sector addresses as an attack vector.

$$index[16] = MD5(salt[0...7] + sect + salt[8...15])$$

The resulting 16 bytes of index are used to "cherry-pick" 16 bytes from the master key, which are run through MD5 with the sector address inserted in the middle: [5]

[5] In both cases the encoded sector address were put in the middle of the MD5 input data only for reasons of symmetry, cryptographically it makes no difference.

$$keykey = MD5($$
$$masterkey[index[0]] + \ldots + masterkey[index[7]] +$$
$$sect +$$
$$masterkey[index[8]] + \ldots + masterkey[index[15]])$$

Some research have cast doubt about the cryptographic qualities of the MD5 algorithm [RSAMD5]. The questionable property, how hard it is to generate collisions, is of no concern here: MD5 is merely used as a "bit blender" and unlike when MD5 is used for authentication purposes, there is no value to the attacker in producing a collision with a different input.

7.5. Sector data encryption

To encrypt and write a sector of plain-text data, the key-key is derived as above, and the sector containing the encrypted sector keys is read into memory.

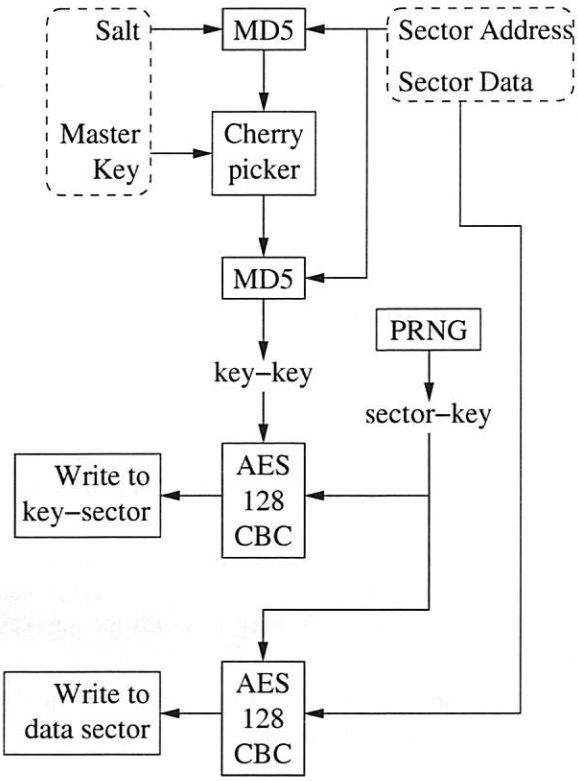

fig 2: write operation

A new sector key is generated with the standard (P)RNG facilities in the kernel and the sector data encrypted using AES/CBC/128.

The sector key is encrypted with AES/CBC/128 using the key-key and inserted at the correct place in the key sector.

Both the key sector and the encrypted data sector are written to disk after which the temporary storage is cleared and the transaction is complete.

To read and decrypt a sector, the key sector and the encrypted sector are read into memory, the key-key generated and the sector key decrypted with it. The sector key is used to decrypt the encrypted sector data, the temporary buffers erased and the transaction completed.

Since the sector data is encrypted with a (pseudo-)random key, there is no need to use any particular initialisation vector for the encryption process and therefore a constant IV of zero bits is used [6].

7.6. Performance optimisations

If implemented strictly according to the above description, GBDE would take a drastic performance hit because all I/O requests, no matter their size would explode into two requests per sector on the underlying device for each sector in the request. Even with the caches built into modern disk drives, this would lead to a drastic performance hit.

Instead the request is split into a number of "work packets" which are characterised by the fact that all data sectors in the work packet are consecutively laid out on the device, this also implies that they are in the same zone and therefore need the same key sector.

Furthermore to avoid re-reading a key sector which has just been written to disk, a small cache is used to keep copies of the encrypted key sectors which have recently been used. The size of the cache grows linearly with usage of key sectors, and decays with 10% every second [7]

Furthermore, the logical sector size of the encrypted device can be set to a power of two multiple of the underlying devices sector size. Using larger sectors means fewer keys, larger zones and consequently larger work packets.

For paging spaces, the VM page size should be used as logical sector size, for file systems the smallest allocation unit should be used. For UFS/FFS this is the so called fragment size, and it is typically 2048 bytes.

In retrospect zero bits were a bad choice since any sector on the encrypted device which contains all zero bits show up as a distinct signature on the encrypted side. This does not introduce a vulnerability inside the design envelope, but it will nonetheless be addressed in the next revision of GBDE since a fix will not affect the installed user base.

[7] It is not possible to use the regular kernel buffer cache because GBDE operates below the SPECFS layer. Furthermore the buffer cache offers no means to clear the buffers contents before it is recycled.

8. Administrative operations

In order to use a device with GBDE it must be initialised first. The initialisation will generate the contents of one lock sector and encode, encrypt and write it to the device.

While this is technically enough to get started, it is highly recommended that the entire device be filled with (pseudo-)random bits before writing the initial lock sector, since this will prevent an attacker from using the previous possibly less random sector contents to eliminate a large fraction of the sectors from an attack.

Unfortunately, it takes considerable amount of time to generate (pseudo-)random bits for modern sized disk devices, and this process can take hours or even days to complete.

To use a GBDE device, it needs to be "attached". This involves presenting the passphrase, and using that locate and decrypt, decode and validate the lock-sector.

The opposite process, detaching the GBDE device is mainly a question of erasing all traces of the lock sector and key sector copies from memory.

Since the location of all lock sectors is recorded in each lock sector, it is possible for any user with a valid passphrase to overwrite any lock sector.

One operation is writing a new lock sector protected by a new passphrase. If done for the lock sector which the user has a passphrase for, this amounts to changing the users passphrase.

Another option is to overwrite the lock sector(s) with zero bits. This disables the passphrase(s), a condition which GBDE will report specifically on, in order to provide the tangible feedback of destroyed data discussed earlier.

9. Attacking GBDE devices

Given a media containing data protected with GBDE, there are two avenues for attacking the algorithm: From the top, trying to get hold of the master-key and salt from one of the lock sectors, from the bottom, trying to derive them from decrypted sectors, or by trying to guess the passphrase.

9.1. Top down attack

The first challenge is to identify which sectors contain what.

There is currently no known algorithms or statistical methods which are able to tell bits produced with AES/CBC/256 and AES/CBC/128 apart, nor is it possible to tell them apart from random bits, so the only way to locate the lock sectors is by doing a brute force search.

If the encrypted location of the lock sectors is available, it may be faster to brute-force the 128 bits which encrypt the location than to search all possible byte offsets on the disk.

Attacking the lock sectors means finding the 256 bit encryption key for the encoded data, and the permutation of the 12 fields in the lock sector.

Given that one of the fields is a MD5 hash which will can be used to test for a hit, the worst case work to brute force the lock sector is therefore somewhere in the neighbourhood of:

$$W_{AES/CBC/128} \cdot 2^{128} + (W_{AES/CBC/256} + W_{MD5}) \cdot 2^{256}$$

or

$$N_{bytes\_on\_device} \cdot (W_{AES/CBC/256} + W_{MD5}) \cdot 2^{256}$$

or, if the attacker by some other means have located the lock sector:

$$(W_{AES/CBC/256} + W_{MD5}) \cdot 2^{256}$$

All of which are practically infinity by todays standards.

9.2. Bottom up attack

Attacking from the other end, so to speak, we will assume that the attacker either knows the plain text of a number of sectors or is able to recognise it by some algorithm, and let us give him the benefit of knowing all the parameters which goes into the mapping of sectors.

Doing a brute force on one of the sectors yields the sector key which can be used to brute-force the encrypted copy of the sector key, which again yields the key-key for this particular sector.

Having obtained the key-key for one specific sector, the next step is to find the 16 unknown bytes of the 24 bytes input sequence to MD5 which generate this key-key as output. (The middle 8 bytes is the little-endian encoded sector number, which we assume the attacker already knows).

In the worst case, the workload so far is:

$$2 \cdot W_{AES/CBC/128} \cdot 2^{128} + W_{MD5} \cdot 2^{128}$$

The attacker now has the value of up to 16 bytes (there may be duplicates) of the 256 bytes in the master key, but he does not know where those 16 values are located in it.

If the attacker brute-forces multiple sectors, he can try to brute-force the 128 bit salt using the known master

key bytes as truth detector. The exact number of sectors he needs to brute force is very hard to predict but let us assume the worst case value of two. The worst case work necessary to do this is:

$$W_{MD5} \cdot 2^{128}$$

Knowing the salt the attacker can now predict which bytes in the master-key are involved in the generation of the key-key for each sector on the disk, and may be able to decrypt a number of sectors based on the subset of the master key he knows so far.

To extend his knowledge of the master key he will have to attack more sectors using brute force, but the search space is reduced by the already known bytes of the master-key so each new sector may be obtainable along with a byte of master key by a worst case effort as low as:

$$2 \cdot W_{AES/CBC/128} \cdot 2^8 + 2 \cdot W_{MD5}$$

at which point the door is wide open.

Summing up, the total lowest worst case effort, given known plaintext is in the neighbourhood of

$$2^{129} \cdot W_{AES/CBC/128} + 2^{128} \cdot W_{MD5}$$

which is also practically infinity by todays standards.

9.3. Attacking the pass-phrase

As already mentioned, using human language text for pass-phrases yields only about one bit of entropy per word, so a well written 64 bit entropy pass-phrase could be:

> Blow, winds, and crack your cheeks! rage! blow!
> You cataracts and hurricanoes, spout
> Till you have drench'd our steeples, drown'd the cocks!
> You sulphurous and thought-executing fires,
> Vaunt-couriers to oak-cleaving thunderbolts,
> Singe my white head! And thou, all-shaking thunder,
> Smite flat the thick rotundity o' the world!
> Crack nature's moulds, and germens spill at once,
> That make ingrateful man!
> [LEAR]

Given that few actors can deliver that correctly over the lime-lights, we can expect that the average pass-phrases will offer less entropy, and dictionary attacks are consequently not only feasible, but to be expected.

The worst case work required to test one passphrase candidate amounts to:

$$W_{SHA2/512} + W_{AES/128/CBC} + W_{disk-read} + W_{AES/256/CBC} + W_{MD5}$$

while the best case work to reject it (because the decrypted location of the lock-sector would be found to be outside the physical media) is only:

$$W_{SHA2/512} + W_{AES/128/CBC}$$

The LOOP-AES [LOOPAES] facility in Linux offers an "iteration count" which adds N thousand iterations of AES/256 to the pass-phrase preprocessing path in order to make such an attack more expensive.

With the increasing speeds of hardware and the availability of cryptographic co-processors, protecting a weak pass-phrase for a decade or more in this fashion is neither realistic nor prudent, and GBDE therefore does not implement a similar option.

A better way to frustrate a dictionary attack is to combine the pass-phrase with a high-entropy token,[8] for instance 1024 bits generated in a suitable random way. This token could be stored on a authentication device or removable storage device such as an USB-key.

10. Known weaknesses

No armour is impenetrable, and no dragon without a soft spot. Three issues have been identified where GBDE has less than perfect performance.

10.1. The static master key

The fact that the lock-sector contents does not change with a change of passphrase means that once a person has once had access to a device, it is not possible to reliably revoke that access by changing the pass-phrase: He would be able to restore a copy of the lock-sector for which he knows the pass-phrase, and thus gain access to the device. All things being equal, this is only marginally more problematic than the fact that the user may also have kept copies of the plain-text data. If this is a real concern, the proper mitigation is to copy the data off the device, reinitialise the GBDE lock sectors, and copy the data back. In the future an "fast re-encrypt" operation would help make this a less painful procedure.

10.2. Huge appetite for random bits

Because sector keys are single use (pseudo-)random bits, GBDE can consume up to sixteen PRNG bytes for every 512 byte sector written to the disk, or 32 kilobytes per megabyte of plaintext. It is obviously important that these PRNG bits are high quality and that the source cannot be manipulated or tampered with, in particular when the device is first initialised where the master-key and salt bit-strings are chosen.

[8] This principle is often referred to as "something you have + something you know"

If the attacker can predict or manipulate the bits assumed by GBDE to have random properties, attacking the protected data could become trivial.

It follows from this that a PRNG which is not regularly perturbed by outside entropy sources would be a very bad source for GBDE, because it would be possible to predict all future bits once the state of the generator was determined once.

In FreeBSD the kernel random facility is based on the industry standard Yarrow algorithm which uses cryptographic functions to churn out high entropy bits based on entropy collected from hardware sources [YARROW].

10.3. The cleaning lady copy attack

An attacker in position to make a bit-wise copy a GBDE protected media on a regular basis would be able to gain a head-start on attacking the device by being able to monitor which sectors change and which do not from one snapshot to the next.

Given just a few copies spaced a day apart, an attacker could likely pinpoint the location of the UFS/FFS super blocks because of their strict geometric distances, and the lock sectors may stand out after a longer time period. In the analysis of attacks on cold media, we more or less assumed that the attacker had this head-start, but the full impact of this kind of attack can only be judged for a specific file system or content type.

A log structured file system would probably be harder to unravel this way because data and metadata are all intermingled in sequential write operations.

11. GBDE as a component

A facility like GBDE is obviously only a component in any real security implementation. The main interface to GBDE as a component is the passphrase and the administrative operations.

11.1. Typical laptop deployment

In an organisation, one possible way to deploy GBDE on laptop computers could be the following:

The laptop is installed with operating system and disk space is set aside for the GBDE partition(s). The security department initialises the GBDE partition with a top-secret company passphrase. The second lock sector is initialised using a passphrase which the branch-office manager has access to, and the third lock sector is initialise with a new passphrase which the user is given access to.

To further improve security, the security department could copy the first two encrypted lock sectors to a floppy disk and replace them with random bits.

Should the user loose his passphrase or if he decides to activate the lock sector destruction function in some situation, the floppy disk can be used to restore the lock sectors and the contents of the device can be recovered from there.

Needless to say, destroying the lock sectors adds little security if the adversary has access to a backup copy, so while the floppy disk does not need by definition need to be stored in a safe, it should obviously not be stored next to the encrypted device either.

11.2. Server deployment

For server use the issue is slightly more complex. It is usually desirable to be able to keep the passphrase stored on the server so that it will be able to boot automatically, but at the same time make it impossible for an attacker to get hold of the passphrase and the protected disk at the same time.

One way to implement this, is by using a "weak link/strong link" method similar to that use in "permissive action links" in atomic weapons [PAL]:

The server is installed in an small enclosure with a very sensitive and very fast intrusion detection. If an intrusion is detected (breach of the weak link), the computer will destroy the passphrase (disabling the strong link), thereby protecting the encrypted data.

Needless to say, there needs to be an internal power source which can provide sufficient power to reliably destroy the passphrase, in case the attacker starts out by cutting the power supply to the enclosure.

12. Performance impact

With an emphasis on strong crypto, performance will always suffer, and GBDE is no exception.

12.1. Throughput

The most expensive part of the GBDE design is the stored encrypted sector key, which worst case forces two disk operations per sector operation.

The first set of measurements try to gauge how much the logical sector size affects the performance. To eliminate as much of the physical aspect, a test of 100 megabytes sequential read and write using 1 megabyte operations were run with varying GBDE logical sector sizes and without GBDE.

The hardware used was an AMD Athlon 700MHz CPU using an IBM DTLA-307015 disk. For workstation or server use, this is somewhat behind the state of the art, but its performance is pretty close to that of a modern laptop computer.

Operations	seq. read		seq. write	
Mode	kB/s	stddev	kB/s	stddev
unencrypted	36141	28	27915	738
512 bytes	7326	12	3447	30
1024 bytes	8088	41	6023	50
2048 bytes	7880	32	5082	25
4096 bytes	8140	176	6061	51
8192 bytes	8849	37	6597	8

As can be see from the numbers, GBDE runs at approx 20-25% of the unencrypted speed if the logical sector size is one kilobyte or above.

Why using 2048 byte logical sectors is slower than both 1024 and 4096 is somewhat of a mystery. We speculate that it may be related to memory management arena fragmentation, or possibly a cache interaction with the UMA memory allocator.

12.2. Latency

Given that the encryption operations are very fast compared to the seek times of contemporary disk hardware, the expected behaviour with respect to I/O latency is that operations with a high degree of locality will take twice as long, and for operations where the disk mechanism incurs arm settling time, the increase in I/O latency will be lost in the noise.

An attempt was made to measure GBDE's impact on I/O latency using a modified version of the `diskinfo -t` tests.

The numbers collected confirmed that highly local operations roughly double their I/O latency, typically from 150µs to 300µs. But for less local operations the measurements were so noisy that statistically they prove nothing useful, one way or the other. A "hairy-eyeball" judgement of the numbers found nothing which would disprove our hypothesis and found them generally compatible with it.

We believe that the noise was due to the complexity of the situation, where GBDE's sector cache, the disk drivers use of the disksort() function and the disk drives internal optimisations result in very significant changes in the order of things for even very minor changes to the driving program.

12.3. Other indications

No attempt to quantify CPU usage more precisely than "will eat a lot of CPU cycles" have been attempted, the expectation is that this will improve when hardware assisted cryptographic functions become available.

The author's personal home directory has been GBDE encrypted for more than 6 months at this point in time, and he has neither found the performance annoyingly slow nor lost any files in this period.

13. Future improvements

The current implementation of GBDE does not take advantage of hardware assisted cryptographic processing. Once AES capable hardware accelerators become available the GBDE workflow engine should be changed to take advantage of them.

A number of additional administrative operations can be imagined, for instance a mode where a device is initialised and attached in one operation which does not write the lock sectors to the media thereby preventing later re-attachment. Such a mode would be useful for devices used for paging or temporary file systems. A command line option to save and restore lock sectors to a file would be convenient too.

Fast re-encryption operation: it is possible to write code to change the master key and salt which only decrypt and re-encrypt the sector-keys. This would be two to three orders of magnitude less work than re-encrypting the entire volume.

14. Availability

GBDE is distributed under the "two-clause BSD license" and will be maintained in the publicly available FreeBSD CVS repository, from where we encourage other operating systems to adopt it.

15. Conclusion

GBDE was designed and implemented as a cryptographic facility which operates on the raw disk level with real-world compatible semantics.

The operational experience so far is good, and initial user feedback has been very positive.

The reviews have so far agreed that the cryptographic design is sound and very strong approching overkill for most "normal" applications.

It is impossible to set a guaranteed protection time on any cryptographic algorithm, but we belive that if AES/128 retains at least 80 bits of effective strength,

GBDE will protect its payload against any terrestial attacker for at least 10 years and probably also 25 years ... provided the attacker can not guess that the passphrase was: "Det er svært at spå, især om fremtiden." [9]

16. Acknowledgements

The author would like to thank:

Robert Watson for organising funding and subsequently translating danglish to the proper red-tape protocol for the involved paper tigers.

Lucky Green for, in addition to being incredibly patient and helpful, trusting his data to GBDE from a very early stage.

David Wagner for his insight, review and in particular for reiterating the obvious until it finally registered.

Bruce D. Evans for his enthusiastic resistance and competent technical hindrance. It would have been much to easy to cut corners if it were not for people like Bruce, much appreciated!

Also thanks to all the people who have listened attentively while I have bored them with incoherent explanations and fuzzy drawings, and people who have helped stamp out "Danglish" from this paper, *Flemming Jacobsen* and *Gregory Sutter* in particular.

And finally a big thanks to the *FreeBSD crew* for putting up with me and my crazy ideas.

17. References

[CGD]
The "cgd" facility in NetBSD: http://netbsd.gw.com/cgi-bin/man-cgi?cgd+4+NetBSD-current

[CPRM]
CPRM was an attempt to mandate Digital Restriction Management features be implemented in all read-write storage devices, such as ATA hard disks.

For a full time line of its emergence and defeat see this summary article on "The Register" 18-02-2001: "CPRM on ATA - Full Coverage" By Andrew Orlowski (http://www.theregister.co.uk):

[DOJ0227]
"The Federal Bureau of Investigation's Control Over Weapons and Laptop Computers"

Report No. 02-27, August 2002, Office of the Inspector General reports 317 laptops lost over 28 months. This is about 2% of the FBI's inventory, and a rate of one every three days.

[IRAN]
"Held Hostage in Iran - A First Tour Like No Other" William J. Daugherty, 1996 http://www.cia.gov/csi/studies/spring98/iran.html

[LEAR]
Shakespeare: King Lear, Act III, scene 2.

[LOOPAES]
The "loop AES" facility in Linux: http://loop-aes.sourceforge.net/loop-AES.README

[PAL] "Permissive Action Links" Steven M. Bellovin http://www.research.att.com/~smb/nsam-160/pal.html

[PGP]
"Pretty Good Privacy", originally written by Phil Zimmerman, subsequently given the dot-com runaround. A good place to start: http://www.pgpi.com

[PRIV]
Private communication. This policy is confidential, so the source cannot be revealed.

[RSAMD5]
RSA Laboratories' bulletin Number 4 - November 12, 1996 "Recent Results for MD2, MD4, and MD5" ftp://ftp.rsasecurity.com/pub/pdfs/bulletn4.pdf

[STEGFS]
"The Steganographic File System" Paper by Ross Anderson, Roger Needham and Adi Shamir. http://www.mcdonald.org.uk/StegFS/

[THEREG]
See for instance, these articles on "The Register" 24-03-2000: "Sneak thief steals state secrets in MI5 laptop" 06-04-2000: "Third secret-packed official notebook nicked" 18-04-2000: "FBI admits loss of 'top secret' laptop" and on a slightly different theme: 23-08-2001: "$15k reward offered for lost laptop" (http://www.theregister.co.uk):

[YARROW]
Yarrow is a secure pseudorandom number generator designed by Bruce Schneier and John Kelsey. http://www.counterpane.com/yarrow.html

[9] "It is difficult to predict, in particular the future"
— Robert Storm Petersen (1882 - 1949)

Cryptographic Device Support for FreeBSD

Samuel J. Leffler

Errno Consulting
sam@errno.com

ABSTRACT

FreeBSD recently adopted the OpenBSD Cryptographic Framework [Keromytis et al, 2003]. In doing so it was necessary to convert the core framework to function correctly in a fully-preemptive/multiprocessor operating system environment. In addition several issues with the basic design were found to cause significant performance loss. After addressing these issues we found that FreeBSD outperformed OpenBSD on identical hardware by as much as 100% in tests that exercise only the cryptographic framework. These optimizations result in similar performance improvements for facilities like IPsec that make heavy use of the cryptographic framework. We observed that FreeBSD's Fast IPsec [Leffler, 2003] typically outperforms OpenBSD's IPsec implementation [Miltchev et al, 2002] by more than 50% on identical hardware.

We conclude that the OCF cryptographic API can be optimized and re-tuned to deliver substantially better performance than the original OCF implementation with large gains in both throughput and latency. Moreover these changes can be made with no impact on clients of the cryptographic framework: both user and kernel sofware designed for the original OCF is easily ported to the FreeBSD implementation of OCF.

1. Background and Introduction

Cryptographic transformations are an important component of security applications and protocols. Because these operations are computationally expensive vendors have developed products that accelerate the calculations and offload the work from the main processor. The OpenBSD Cryptographic Framework (OCF) [Keromytis et al, 2003] was developed to provide a uniform interface to cryptographic resources. It provides an in-kernel API to cryptographic resources and a device interface for user-level access to hardware-accelerated cryptographic operations. This functionality is critical for high performance implementations of security protocols such as IPsec [Kent & Atkinson, 1998], SSL [Freier et al, 1996], TLS [Dierks & Allen, 1999], DNSSEC [Eastlake, 1999], ssh [Ylonen et al, 2002], and Kerberos [Kohl & Neuman, 1993]. Good cryptographic support is also important for implementing encrypted secondary storage [Gattaneo et al, 2001] and virtual memory [Provos, 2000].

Recognizing the importance of this facility, FreeBSD recently adopted the OCF. Porting the software however required addressing several issues.

- The OCF was designed for a uniprocessor system without kernel preemption. The FreeBSD 5.0 operating system has fine-grained locking and the kernel is fully preemptive. This required a rewrite of the core crypto functionality.
- The I/O framework used on OpenBSD is different than that used by FreeBSD. The crypto device drivers required significant modification to deal with these differences.
- The OCF duplicates cryptographic support already present in FreeBSD for the KAME IPsec software. In some cases the existing implementations are superior to those provided by the OCF. Rather than duplicate this software each component was evaluated and the best was chosen.

After addressing these issues we looked at the performance and found that it was suboptimal. [Keromytis et al, 2003] states the "OCF attains 95% of the theoretical peak device performance." These results however are for relatively slow devices and fail to consider the overhead required to reach that performance. Furthermore, the 95% figure is misleading in that it is attained for a case that rarely occurs and where the majority of the overhead of the OCF is hidden by the raw computational cost.

In analyzing the performance of OCF we found several problems that contribute to suboptimal performance for all operations:

1) The OCF requires two kernel thread context switches for each operation. On many systems the peak performance of the OCF is limited by the rate at which the system can do context switches.
2) The OCF forces all crypto operations to pass twice through a single kernel thread. Combining the dispatch and return processing in a single thread increases latency.
3) Crypto device drivers trade off throughput for latency. This has a detrimental effect on the performance of network protocols where latency is critical.

Furthermore it was observed that when crypto devices are overloaded the OCF reacts by discarding operations instead of applying flow-control techniques. This results in severe performance loss for network protocols as this action is equivalent to discarding packets.

With these problems corrected FreeBSD was found to outperform OpenBSD on identical hardware by as much as 100% in tests that exercise only the cryptographic framework. The overhead of using the cryptographic facilities was dramatically reduced raising the peak performance and making more of the CPU available for non-cryptographic work. These optimizations result in similar performance improvements for facilities like IPsec that make heavy use of the cryptographic framework. For example, FreeBSD's Fast IPsec [Leffler, 2003] outperforms OpenBSD's IPsec implementation [Miltchev et al, 2002] by more than 50% on identical hardware running **netperf** over a 3DES+SHA1 tunnel.

The remainder of this paper is organized as follows. Section 2 describes the cryptographic framework and discusses the effort needed to make it operate in a fully preemptive environment. Section 3 discusses performance issues, starting with the tools used to evaluate performance and progressing through each of the issues that were identified in the software. Section 4 describes performance results for FreeBSD and compares them to OpenBSD. Section 5 outlines the the status and availability of this work and talks about future work. Section 6 gives conclusions.

2. Porting the Cryptographic Framework to FreeBSD 5.0

The OCF is comprised of three components:

1) A core set of code that manages a registry of crypto device drivers, dispatches crypto operations to drivers, and coordinates the return of results from drivers to the submitter.
2) Crypto device drivers that submit crypto operations to hardware devices and return results to the crypto core.
3) The **/dev/crypto** pseudo-device driver that provides linkage between user-level software and the core crypto support.

The core crypto support and the **/dev/crypto** driver are simple pieces of software. The crypto device drivers however represent a significant development effort. Some aspects of cryptographic device hardware that vendors refuse to disclose have been reverse-engineered and this work is embodied in these drivers. In considering the integration of the OCF into FreeBSD it was important to maintain the driver API so that existing drivers could be easily reused and so that ongoing development could be shared by both OpenBSD and FreeBSD communities.

The initial port of the OCF to FreeBSD 5.0 was reasonably straightforward. Instead of using processor priority to insure critical code are not executed concurrently, locking primitives were used to guard concurrent access to data structures. This resulted in certain constructs being recast and some code being rewritten. For example, the OCF blocks concurrent access to all its data structures by raising the processor priority to **splimp**. This effectively blocks all concurrent activity in the lower half of the kernel from reentering the code. In FreeBSD 5.0 however splimp does nothing; instead concurrency primitives such as a **mutex** must be used. But simply replacing splimp usage with a single mutex does not work unless the mutex semantics are relaxed to permit recursive acquisition. Instead each of the data structures used by the cryptographic framework were given their own mutex and some code was reorganized to insure consistent lock ordering is used to avoid deadlocks. The end result is easier to understand and permits greater concurrency. The only downside is that mutex primitives have more overhead than simply manipulating the processor priority level. In practice this additional overhead is swamped by other costs and not critical to overall performance.

3. Performance Analysis

This section briefly describes the tools and techniques used to evaluate performance and then discusses the performance problems that were identified.

3.1. Tools

OpenBSD provides no statistics or other facilities for understanding the performance of the OCF. The only tools for evaluating performance treat the system as a "black box"; e.g. the openssl tool from the OpenSSL distribution.

We began by instrumenting the core software and each crypto device driver. Every error or failure is accounted for separately so problems can be determined immediately. (This is especially important for understanding problems reported by naive users.)

Next, tools were developed to exercise the crypto functionality. One such tool is the **cryptotest** program that submits symmetric-key crypto operations through the **/dev/crypto** device and verifies the results. cryptotest measures performance over a range of operations and parameters and is a basic tool for evaluating software changes and hardware configurations[1].

Finally a profiling facility was added that timestamps crypto requests as they pass through the system. At important points in the processing of crypto requests these timestamps are used to calculate minimum, maximum, and average time spent doing each task. The cryptotest program has a −p option that enables collection of profiling statistics over the duration of a run.

	Time (ns)			
Task	Avg	Min	Max	Description
Dispatch	115	114	640	Delay before op is handed to driver.
Invoke	155883	154143	193937	Delay before op is returned by driver.
Done	2295	2196	18342	Delay before callback method is invoked.
Callback	341	323	733	Duration of callback method processing.

Table 1: *Sample crypto profiling results*

Table 1 shows the results from running cryptotest on a system with a Broadcom BCM5822 accelerator card. The machine has an Asus P4B533-V Intel845G motherboard with 1.8 GHz P4 processor. The Broadcom card was in a 32-bit/33MHz PCI slot. The tests performed 3DES calculations on 8192-byte buffers. Four values are calculated; one for each of the tasks done by the core crypto framework. The **Invoke** value is especially interesting as it gives an approximation of the peak processing capability of the hardware (modulo the overhead of the device driver). In this case 8 kilobytes were processed in an average of 156 milliseconds for a peak bandwidth of 410 Mb/s. Profiling adds a fixed overhead to each operation; reducing performance results by 2-8%. This facility is described in more detail in Section 4.

[1] The cryptotest program described here is derived from code provided by Theo de Raadt.

3.2. Problems

The normal flow of crypto operations in the OCF is as follows:

1) Client formulates a crypto operation and submits it through the "crypto_dispatch" routine. This routine places the request on a "dispatch queue" and notifies a kernel thread.
2) The kernel thread removes the operation from the dispatch queue and calls "crypto_invoke" to hand the request off to the appropriate crypto device driver.
3) The crypto device driver processes the operation. Typically this is done by submitting one or more commands to the hardware device. Some drivers however may process the data immediately; e.g. the software crypto driver.
4) When the crypto request is completed the crypto driver calls the "crypto_done" routine to notify the crypto subsystem the request is done. This routine places the request on a "return queue" and notifies a kernel thread.
5) The kernel thread removes the operation from the return queue and invokes the callback method associated with the request.

This scheme requires two kernel thread context switches to process each cryptographic request. The trip through the kernel for dispatch is done so that batching can be carried out [Keromytis, 2003]; except the OCF does not support batching. The second trip through the kernel thread for returning requests is done because the callback methods typically take a long time to run. By moving this work from the device driver (typically in the interrupt service routine) this long-running work can be done at a low interrupt priority level. The problem with this is that *all* callback methods are handled in this way, so even those methods that do very little incur this overhead.

For many systems the peak performance of the OCF is limited by number of context switches the system can perform. Reducing this overhead is therefore important.

3.2.1. Separating Dispatch and Return of Results

The first observation was that doing dispatch and return processing in a single kernel thread was suboptimal. While the two tasks were interleaved this still introduced latency as return processing is potentially very time consuming and could block the dispatch of operations to the hardware. Splitting the work into separate threads eliminated contention for shared resources and reduced latency, especially under FreeBSD 5.0 where the kernel is preemptive. Furthermore, under FreeBSD 5.0 doing both tasks in a single thread required that access to both the dispatch and return queues be synchronized with one lock which increased contention further. When dispatch and return processing was split into separate threads IPsec performance increased by 10-15%.

3.2.2. Eliminating Context Switch Overhead

Next it was observed that dispatch processing does not need to be done in a kernel thread [Stone, 2002]. Replacing the kernel thread with a software interrupt thread reduces the cost to enter the dispatch loop by more than 50% on FreeBSD 4.8.

	Dispatch Time (ns)		
	SWI	Thread	Speedup
Avg	427	1519	3.6x
Min	380	1292	
Max	9248	12585	

Table 2: *Software interrupt- vs thread-based dispatch*

Table 2 shows measurements of the dispatch time collected with cryptotest and profiling on the Asus-based test machine. We also tried to convert the return processing thread to a software interrupt but this failed because the callback methods were not prepared to be entered at an elevated priority level.

The main difficulty with converting the dispatch thread from a kernel thread to a software interrupt thread was managing the interrupt masks and synchronization with other threads of execution in the kernel. A new "spl", **splcrypto**, was defined that blocks both the software interrupt thread, crypto hardware interrupts, and all networking software interrupts. This is used within the core crypto code to guard access to data structures that are used in the software interrupt thread.

After converting the dispatch procedure to a software interrupt thread it was observed that most operations can be dispatched without a context switch at all! Recall that the reason for switching to another thread is to have a common location to batch operations. However most crypto requests should not be batched because batching them increases latency, reducing overall performance. This is especially true of network protocols such as IPsec.

To handle this conflict between a need for low latency and a desire for high throughput we introduced the notion of **batchable** crypto requests. Requests that are marked batchable are queued and dispatched from the dispatch thread that is entered through a software interrupt. Crypto drivers are supplied a hint at step 2 in the above procedure that indicates whether more operations will follow immediately; this permits drivers to safely batch operations together. Operations that are not marked batchable are sent to the crypto drivers immediately without passing through the dispatch thread. Doing this requires no changes to the dispatch thread.

	Dispatch Time (ns)		
	Direct	SWI	Speedup
Avg	108	427	4x
Min	99	380	
Max	1634	9248	

Table 3: *Direct- vs software interrupt--based dispatch*

As Table 3 shows, direct dispatch lowers dispatch overhead dramatically. In addition, variance (not shown) is significantly reduced which is important for clients like network protocols.

While the above techniques eliminate the context switch needed to dispatch operations they do not eliminate the context switch prior to invoking the callback method. As discussed earlier, the callback method may take a long time to execute so typically should not be run directly from the crypto device driver. Some callback methods however execute quickly. In particular the callback method for the **/dev/crypto** device driver does little more than wakeup the thread that submitted the request. This callback can safely be called from the device driver without switching to the crypto return thread. Crypto requests that want their callback methods invoked directly mark their request appropriately. This eliminates the context switch normally required to return results. Clients however must be careful when submitting requests as the results may be ready before they can wait for them. (To aid in recognizing this, the crypto framework now marks crypto operations **done** on completion so clients can submit an operation, synchronize access to the operation data structure, and then verify a request is incomplete before blocking to wait for a result.) With this change operations submitted through **/dev/crypto** can be done without any context switches! Table 4 compares the cost of using

immediate callbacks to passing through the return thread.

	Return Time (ns)		
	Immediate	Task	Speedup
Avg	98	3102	33x
Min	97	3010	
Max	455	15098	

Table 4: *Immediate- vs task--based returns*

Similar to the above technique, when a crypto device driver operates synchronously (e.g. the software crypto driver), passing callbacks through the return thread is typically unnecessary. When a caller is prepared for immediate callback with a result it can mark operations appropriately and bypass the trip through the return thread. This last optimization effectively returns the overhead of using software crypto support to the cost of a function call. The only downside to this technique is that because the callback is done in a continuation style there are two extra stack frames required to process a request to completion, which may be significant for a kernel that is running close to the limit of its kernel stack size.

Note that all these optimizations reduce the *overhead* of using the crypto subsystem. This is most noticeable for operations with small operands as the cost to do the cryptographic operation is small compared to the overhead to process it. As operand size grows the time spent computing the result reduces the importance of the optimizations described here. However since operand size typically exhibits a bi- or tri-modal distribution these optimizations have a significant impact on real-life performance.

3.2.3. Tuning Drivers to Minimize Latency

Two crypto device drivers were imported from OpenBSD when this work was brought into FreeBSD. These drivers support the hardware devices most readily available: those based on the Hifn 7951, Hifn 7811 and Broadcom 58xx. Both drivers attempt to batch symmetric crypto operations when possible. Typically this is possible only when the device is under load. It was observed however that this batching has a severe performance penalty for applications like IPsec because it increases the latency for most operations. The drivers have been changed to batch operations only if they are explicitly marked batchable.

3.2.4. Flow Control for Cryptographic Operations

The OCF assumes that once an operation is sent to a crypto device driver for processing it will either succeed or fail permanently. Failures due to lack of resources are treated as permanent failures. Callers may interpret the error code for failed operations and resubmit them but this is never done. This has ramifications for clients of the OCF. For IPsec, for example, a failed crypto request is equivalent to dropping a packet. This can cause higher level protocols such as TCP to initiate backoff algorithms and for performance to drop.

In many instances drivers can quickly recover from conditions that are reported to the OCF as errors. For example, the Hifn 7951 has a small ring of descriptors that may be exhausted when under load. Since the ring is known to be full when an error occurs one can assume an operation will complete soon and space will be available to submit a new operation. Rather than fail a request because there are no descriptors it is more efficient to queue it and "push back" on the crypto dispatch logic until resources become free. While this can cause excessive queueing of requests, in practice, this does not happen because other flow control mechanisms come into play (e.g. network traffic stops flowing).

Crypto device drivers may return an **ERESTART** error when given a request they cannot process. This should be used only when there is a shortlived resource exhaustion. The crypto framework will queue the request and mark the driver as "blocked" so subsequent requests will be queued and not submitted to the driver. The driver must determine when it is ready to accept requests again and call "crypto_unblock" to clear the blocked condition on the driver. The **ubsec** (Broadcom) and **hifn** (Hifn 7951, 7811, etc.) drivers both implement this flow control scheme. Statistics indicate this condition happens frequently for the Hifn 7951 when under load. To demonstrate this the cryptotest program was run with 28 threads concurrently submitting operations to a single 7951-based card (a Soekris vpn1201 PCI card installed in the Asus-based test machine).[2] Over the course of the test run 51% of the requests required restarting because of temporary resource exhaustion (no space in the command/result rings). Without this flow control mechanism these operations would have failed. If the operations were submitted on behalf of IPsec they would have caused packets to be dropped.

[2] Note that the number of threads had to be significantly inflated because the test was run with an optimized system. This behaviour was initially encountered with an unoptimized system running IPsec under a three-client load.

4. Performance Results

To evaluate the changes described in this paper numerous tests were conducted. The results presented here were mostly collected using the cryptotest tool described above. A few results from higher-level applications such as IPsec are also presented; a more in-depth discussion of those results is found in [Leffler, 2003]. Note that while cryptotest was used to collect most of the results presented here, the data were validated using other tools such the openssl program. After more than a year of testing and tuning cryptotest is considered a reliable indicator of system performance.

Table 5 shows results from running cryptotest on the Asus-based test machine running FreeBSD 4.8 with a variety of hardware devices, while Table 6 shows the results using OpenBSD 3.3 (release) on the same test machine and cryptographic hardware devices.[3]

Operand	7951			7811			5822		
Size	3DES	MD5	SHA1	3DES	MD5	SHA1	3DES	MD5	SHA1
8	1.9	1.9	1.5	1.9	1.9	1.9	3.0	2.9	2.9
16	3.8	3.8	3.0	3.9	3.8	3.8	6.0	5.9	5.8
32	7.6	7.6	6.0	7.6	7.6	7.7	11.7	11.6	11.5
64	14.8	12.0	11.9	15.3	15.2	15.1	22.6	22.4	22.1
128	23.1	22.2	18.9	28.5	28.3	27.0	41.4	42.9	42.2
256	36.2	35.2	31.0	49.7	50.9	41.8	74.9	80.1	78.7
512	51.2	54.2	46.3	75.7	72.3	72.3	128.0	141.5	138.9
1024	63.1	69.8	59.5	99.3	109.0	104.7	205.5	235.1	232.1
2048	71.3	79.2	69.5	120.6	143.9	134.7	282.1	348.1	344.5
4096	76.4	88.2	75.8	136.5	170.9	157.3	346.9	457.2	454.4
8192	78.4	93.5	79.2	145.1	188.7	170.9	380.2	531.9	530.0

Table 5: *Cryptotest results for FreeBSD 4.8*

The OpenBSD results are comparable to where FreeBSD was before any of the optimizations described in this paper were done.

Table 7 directly compares the FreeBSD and OpenBSD results for 3DES on the three devices. As the operand size increases the processing time of the computation reduces the effect of the lower submission overhead. For faster hardware however this is not true because the host is not fully utilizing the cryptographic hardware. For operand sizes less than 2048 bytes FreeBSD outperforms OpenBSD by at least 70% for the fastest hardware device and typically it outperforms OpenBSD by at least 50%.

[3] OpenBSD 3.3 does not work correctly on the test machine with the Broadcom BCM5822 unless APM support is disabled. Before this was discovered each cryptographic operation took about 1 second to complete! However, even with this workaround hardware-assisted IPsec does not function correctly.

Operand	7951			7811			5822		
Size	3DES	MD5	SHA1	3DES	MD5	SHA1	3DES	MD5	SHA1
8	1.2	1.1	1.1	1.2	1.2	1.1	1.5	1.3	1.5
16	2.5	1.9	2.1	2.2	2.4	2.3	3.1	3.0	2.2
32	4.6	3.9	4.2	4.6	4.8	4.0	6.1	6.0	5.3
64	9.2	8.3	7.5	9.1	8.7	7.6	11.8	11.8	11.7
128	16.2	12.8	14.0	18.6	16.6	15.3	22.7	22.0	20.6
256	27.3	24.1	24.0	32.9	30.8	27.6	43.0	40.5	41.2
512	41.5	40.5	38.1	53.9	52.9	54.1	72.9	77.6	80.6
1024	54.0	57.5	52.3	80.4	87.3	84.3	119.1	147.7	127.3
2048	65.9	73.0	64.3	106.5	123.3	116.5	199.6	241.3	239.8
4096	71.5	84.1	72.5	124.1	149.6	142.3	282.3	356.6	355.0
8192	76.8	91.1	77.3	138.9	176.2	161.9	333.1	436.8	459.6

Table 6: *Cryptotest results for Openbsd 3.3 (release)*

Operand	7951			7811			5822		
Size	OBSD	FBSD	%diff	OBSD	FBSD	%diff	OBSD	FBSD	%diff
8	1.2	1.9	58	1.2	1.9	58	1.5	3.0	100
16	2.5	3.8	52	2.4	3.9	62	3.1	6.0	93
32	4.6	7.6	65	4.6	7.6	65	6.1	11.7	91
64	9.2	14.8	60	9.1	15.3	68	11.8	22.6	91
128	16.2	23.1	42	18.6	28.5	53	22.7	41.4	82
256	27.3	36.2	32	32.9	49.7	51	43.0	74.9	74
512	41.5	51.2	23	53.9	75.7	40	72.9	128.0	75
1024	54.0	63.1	16	80.4	99.3	23	119.1	205.5	72
2048	65.9	71.3	8	106.5	120.6	13	199.6	282.1	41
4096	71.5	76.4	6	124.1	136.5	9	282.3	346.9	22
8192	76.8	78.4	2	138.9	145.1	4	333.1	380.2	14

Table 7: *3DES performance comparison on Asus P4B433-V*

To understand the importance of these optimizations on slower systems the cryptotest measurements were repeated on a Dell XPS 300 system (300 MHz Intel PII processor and 64 Mbytes of memory) using each of the hardware crypto devices. Again FreeBSD 4.8 and OpenBSD 3.3 were used.

Operand	7951			7811			5822		
Size	OBSD	FBSD	%diff	OBSD	FBSD	%diff	OBSD	FBSD	%diff
8	0.7	1.1	57	0.7	1.1	57	0.9	1.1	22
16	1.5	2.1	40	1.5	2.1	40	1.7	2.3	35
32	3.0	3.9	30	3.0	4.3	43	3.4	4.5	32
64	5.5	7.4	34	5.9	8.0	35	6.6	8.9	34
128	10.4	13.6	30	11.6	14.7	26	12.6	17.6	39
256	18.1	22.6	24	19.9	26.5	33	24.1	32.6	35
512	28.9	35.2	21	34.6	44.0	27	43.3	59.3	36
1024	40.2	46.6	15	52.9	64.2	21	71.3	93.4	30
2048	51.3	56.8	10	73.3	86.3	17	106.2	134.1	26
4096	60.2	62.8	4	92.8	100.1	7	149.5	169.7	13
8192	65.1	66.7	2	105.8	110.0	3	178.8	191.7	7

Table 8: *3DES performance comparison on Dell XPS 300*

As Table 8 shows, the performance difference between the two systems is less significant. This

appears to be due to the higher overhead to setup I/O operations (e.g. slower bus) reducing the direct benefit of using an outboard cryptographic accelerator. There are also some anomalous results that may be caused by inaccuracy in the timing data (as before, it was necessary to disable APM support in OpenBSD to get it to process cryptographic operations in a reasonable manner).

Finally, Table 9 shows the result of running the **netperf** performance testing tool [Jones, 2003] between the Asus-based test machine and a second machine with a Tyan S2707G2N motherboard and a 1.8 GHz P4 processor. Both machines used dedicated Intel gigabit Ethernet NICs; the Asus machine had an 82540 NIC located in a 32-bit PCI slot, while the Tyan system used an 82545 NIC with a 64-bit PCI interface. Both systems used Broadcom BCM5822 cards: on the Asus machine the crypto cards were in a 32-bit PCI slot; on the Tyan machine the cards were in a 64-bit PCI slot. The systems were physically connected by a cross-over cable. First the netperf tcp_stream_script script was used to collect data on an unencrypted connection for a variety of socket sizes. Next, a 3DES+SHA1 IPsec tunnel was setup using static keying and the same tests were rerun with cryptographic calculations done in software on the host or by the 5822. The figures in the table were reported with a 99% confidence interval (as determined by netperf). Note that no comparable results for OpenBSD were available for the 5822 because the tests failed to complete.

System	Sender	MsgSize	Throughput (10^6bits/sec)		
			Raw	S/W	5822
OBSD	Asus	4096	541	27	-
FBSD	Asus	4096	624	42	160
OBSD	Asus	8192	530	27	-
FBSD	Asus	8192	624	42	160
OBSD	Asus	32768	519	27	-
FBSD	Asus	32768	627	42	159
OBSD	Tyan	4096	648	27	-
FBSD	Tyan	4096	797	44	170
OBSD	Tyan	8192	650	27	-
FBSD	Tyan	8192	798	44	170
OBSD	Tyan	32768	653	27	-
FBSD	Tyan	32768	799	44	169

Table 9: *Netperf test results*

Interpreting these results without careful inspection of the way these two IPsec implementations work is difficult. Nonetheless some useful information can be gleaned.

First, remember that the Tyan-based system had both NIC and Broadcom devices in 64-bit PCI slots while the Asus-based system has only 32-bit PCI support. This difference is noticeable in the netperf results and also when running non-network tests like cryptotest.

In lieu of results for the 5822, the software-based crypto results are the most useful in comparing performance of the crypto subsystems. FreeBSD is 55% faster than OpenBSD on the Asus machine and 62% faster on the Tyan machine. Remember that OpenBSD requires two context switches for each crypto operation; this accounts for much of the performance difference. Otherwise the different results between the two machines is due to the 32- versus 64-bit PCI NIC for the sender.

Netperf results tend to be dominated by the performance of the sender. When using the 5822 the FreeBSD system was typically about 30% idle on the sender and 40% idle on the receiver (according to the vmstat program). Statistics maintained by the FreeBSD Broadcom driver show that on the slower system (the Asus-based machine) a peak of 9 crypto operations were pending completion by the hardware and 20 operations pending in the system (queued waiting for the hardware to have room to accept another operation). On the Tyan-based system these numbers were 6 and 9. Furthermore, 78% of the crypto requests processed by the sender found the hardware device busy and unable to accept a new request. Together these numbers indicate the performance was limited by the crypto hardware.

Running netperf end-to-end over an IPsec connection does not show the full possible performance because the processing is heavily influenced by the location of the sending process. For example, on the sender the data originates in a TCP socket and so must be copied for transmission. Further, when forming an IPsec packet on the sender multiple mbufs are typically created requiring the crypto driver to process fragmented operand data. By contrast, when a system is operating as an endpoint of an IPsec gateway, packets do not originate or terminate on the machine. Instead IPsec packets are received, processed, and then forwarded to their destination. In this configuration the data is almost always contiguous in memory and does not require copying for potential retransmission. Consequently load on the crypto hardware is very different, and utilization is typically 100%.

Finally, it has been observed that to fully utilize fast crypto hardware such as the Broadcom 5822, CPU performance (for Intel-based systems at least) is important. Results for the netperf test described above scale almost directly to the clock speed of Intel P4 and Xeon™ processors up to 2.53 GHz (the

maximum clock speed that was available at the time these tests were done). With 2.4GHz processors end-to-end netperf results with 5822 support exceed 200 Mb/s.

5. Status and Future Work

The initial port of the OCF to FreeBSD was done in September 2001. Integration of this work into FreeBSD was completed November 2002 and committed to the stable branch a month later. The work described here is freely available as part of the FreeBSD 5.0 and 4.8 releases. It is currently being integrated into the NetBSD operating system [Stone, 2003]. Several vendors have incorporated the framework in commercial products.

The techniques described in this paper have not been applied to asymmetric crypto operations. Applying them is straightforward and should yield significant performance improvements.

Otherwise, future work falls into two broad categories: improved device support and load balancing. The current software supports only a few hardware devices and some of these have underwhelming performance or are too expensive and/or too difficult for the average consumer to obtain. The main impediment here is the non-disclosure of programming information. New products are expected that will address both of these concerns.

Otherwise, the most significant deficiency in the current framework is the inability to use protocol-specific hardware operations. Most vendors of crypto hardware optimize their products for use as "all-in-one" devices that take an IPsec/SSL/TLS packet and parse the protocol and perform the cryptographic transforms in a single request. This is incompatible with the current general-purpose API provided by the OCF. Supporting these kinds of operations requires exposing state that is currently private to the protocols. However adding this is the only way that some hardware devices can be used at all as they do not otherwise provide access to the cryptographic transformation hardware.

Load balancing refers to efficiently supporting multiple crypto devices in a single system. These devices may be add-in devices or dedicated CPU's in a multiprocessor system. The OCF does very little in this regard; when multiple devices are present in a system it will assign them in a round-robin fashion when creating **sessions** for symmetric crypto operations. Asymmetric operations do not consider multiple devices; they are dispatched to the first available device that can handle the request. In addition it has been understood for a while that the setup overhead for communicating with hardware devices can be too high to justify submitting small data buffers. Instead such operations should be handled by the main CPU. All these issues requires similar functionality. Experimental work is ongoing to calculate operation-specific cost metrics for cryptographic devices and then use that to dynamically schedule operations as they are submitted. This scheme handles symmetric and asymmetric operations and includes the software crypto device driver so that, for example, operations on small data buffers can be processed in software when appropriate.

6. Conclusions

FreeBSD has imported the OpenBSD Cryptographic Framework. In the process the core code has been rewritten to work correctly in a preemptive environment. We have added statistics and developed tools to aid in understanding system performance and to help in the event of problems. Several deficiencies in the OCF have been identified and corrected. These changes result in a system that performs as much as 100% faster than the OCF on identical hardware. For facilities like IPsec that make heavy use of the cryptographic framework, these changes can improve performance by more than 50%.

We conclude that the OCF cryptographic API introduced by [Keromytis et al, 2003] can be optimized and re-tuned to deliver substantially better performance than the original OCF implementation with large gains in both throughput and latency. Moreover these changes can be made with no impact on clients of the cryptographic framework: both user and kernel software designed for the original OCF is easily ported to the FreeBSD implementation of OCF.

7. Acknowledgements

This work is derived from the OpenBSD cryptographic framework; without it this work would not have been possible. Jason Wright and Angelos Keromytis of were especially helpful in understanding certain design decisions in the OCF. Theo de Raadt provided the original code from which cryptotest was created. Jonathan Stone first suggested that there was excessive overhead in the OCF; this motivated me to attack the problem.

I am especially grateful to Vernier Networks for funding much of this work and providing access to their equipment. Global Technologies Group, Inc. (GTGI) provided two Xl-Crypto (Hifn 7811) boards and funded FreeBSD device support for their cards. Soren

Kristenson of Soekris Engineering provided two vpn1201 cards (Hifn 7951).

References

Dierks & Allen, 1999.
T. Dierks & C. Allen, "The TLS Protocol Version 1.0," *RFC 2246* (January 1999).

Eastlake, 1999.
D. Eastlake, "Domain Name System Security Extensions," *RFC 2535* (March 1999).

Freier et al, 1996.
A. Freier, P. Karlton, & P. Kocher, "The SSL Protocol Version 3.0," *http://home.netscape.com/eng/ssl3/draft302.txt* (November 1996).

Gattaneo et al, 2001.
G. Gattaneo, L. Catuogno, A. Del Sorbo, & P. Persiano, "The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX," *FREENIX Track: Usenix Annual Technical Conference* (June 2001).

Jones, 2003.
R. Jones, "Netperf 2.2pl3: network performance evaluation tool," *http://www.netperf.org* (February 11, 2003).

Kent & Atkinson, 1998.
S. Kent & R. Atkinson, "Security Architecture for the Internet Protocol," *RFC 2401* (August 1998).

Keromytis, 2003.
A. Keromytis, *Private email* (January 2003).

Keromytis et al, 2003.
A. Keromytis, J. Wright, & T. de Raadt, "The Design of the OpenBSD Cryptographic Framework," *USENIX Annual Technical Conference* (June 2003).

Kohl & Neuman, 1993.
J. Kohl & C. Neuman, "The Kerberos Network Authentication Service (V5)," *RFC 1510* (September 1993).

Leffler, 2003.
S. Leffler, "Fast IPsec: A High Performance IPsec Implementation for UNIX Systems," *BSDCon 2003* (September 2003).

Miltchev et al, 2002.
S. Miltchev, S. Ioannidis, & A. Keromytis, "A Study of the Relative Costs of Network Security Protocols," *FREENIX Track: Usenix Annual Technical Conference,* p. 41–48 (June 2002).

Provos, 2000.
N. Provos, "Encrypting Virtual Memory," *Proceeding of the USENIX Security Symposium* (August 2000).

Stone, 2002.
J. Stone, *Private email: Re: anyone working on crypto processor support?* (November 2002).

Stone, 2003.
J. Stone, *Private email: preliminary port of sys/opencrypto to NetBSD* (March 2003).

Ylonen et al, 2002.
T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, & S. Lehtinen, "SSH Protocol Architecture," *Internet WIP draft-ietf-secsh-architecture-12.txt* (January 2002).

Biography

Sam Leffler has been actively working with UNIX since 1975 when he first encountered it at Case Western Reserve University. While working for the Computer Systems Research Group (CSRG) at the University of California at Berkeley he helped with the 4.1BSD release and was responsible for the release of 4.2BSD. He has contributed to almost every aspect of BSD systems; most recently working (again) on the networking subsystem. You can contact him via email at <sam@errno.com>.

Enhancements to the Fast Filesystem To Support Multi-Terabyte Storage Systems

Marshall Kirk McKusick
Author and Consultant

Abstract

This paper describes a new version of the fast filesystem, UFS2, designed to run on multi-terabyte storage systems. It gives the motivation behind coming up with a new on-disk format rather than trying to continue enhancing the existing fast-filesystem format. It describes the new features and capabilities in UFS2 including extended attributes, new and higher resolution time stamps, dynamically allocated inodes, and an expanded boot block area. It also describes the features and capabilities that were considered but rejected giving the reasons for their rejection. Next it covers changes that were made to the soft update code to support the new capabilities and to enable it to work more smoothly with existing filesystems. The paper covers enhancements made to support live dumps and changes made to filesystem snapshots needed to avoid deadlocks and to enable them to work efficiently with multi-terabyte filesystems. Similarly, it describes changes that needed to be made to the filesystem check program to work with large filesystems. The paper gives some comments about performance, and decribes areas for future work including an extent-based allocation mechanism and indexed directory structures. The paper concludes with current status and availability of UFS2.

1. Background and Introduction

Traditionally, the BSD fast filesystem (which we shall refer to in this paper as UFS1) [McKusick et al, 1996; McKusick, Joy et al, 1984] and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. The UFS1 filesystem was designed in the early 1980's when the largest disks were 330 megabytes. There was debate at the time whether it was worth squandering 32-bits per block pointer rather than using the 24-bit block pointers of the filesystem that it replaced. Luckily the futurist view prevailed and the design used 32-bit block pointers. Over the twenty years that it has been deployed, storage systems have grown to hold over a terabyte of data. Depending on the block size configuration, the 32-bit block pointers of UFS1 run out of space in the 1 to 4 terabyte range. While some stop-gap measures can be used to extend the maximum size storage systems supported by UFS1, by 2002 it became clear that the only long-term solution was to use 64-bit block pointers. Thus, we decided to build a new filesystem, UFS2, that would use 64-bit block pointers.

We considered the alternatives between trying to make incremental changes to the existing UFS1 filesystem versus importing another existing filesystem such as XFS [Sweeney et al, 1996], or ReiserFS [Reiser, 2001]. We also considered writing a new filesystem from scratch so that we could take advantage of recent filesystem research and experience. We chose to extend the UFS1 filesystem as this approach allowed us to reuse most of the existing UFS1 code base. The benefits of this decision were that UFS2 was developed and deployed quickly, it became stable and reliable rapidly, and the same code base could be used to support both UFS1 and UFS2 filesystem formats. Over 90 percent of the code base is shared, thus bug fixes and feature or performance enhancements usually apply to both filesystem formats.

Sections 2, 3, and 4 discuss the UFS2 filesystem itself. Sections 5 and 6 discuss enhancements that were made during the development of UFS2 but which transfer over to UFS1 as well. Sections 7 and 8 describe how we overcame problems of scale brought on by the enormous size of filesystems supported by UFS2. The last three sections conclude with discussions of performance, future work, and current status.

2. The UFS2 Filesystem

The on-disk inodes used by UFS1 are 128-bytes in size and have only two unused 32-bit fields. It would not be possible to convert to 64-bit block pointers without reducing the number of direct block pointers from twelve to five. Doing so would dramatically increase the amount of wasted space as only direct block pointers can reference fragments. So, the only viable alternative is to increase the size of the on-disk inode to 256 bytes.

Once one is committed to changing to a new on-disk format for the inodes, it is possible to include other inode-related changes that were not possible within the constraints of the old inodes. While it is tempting to throw in everything that has ever been suggested over the last twenty years, we feel that it is best to limit the addition of new capabilities to those that are likely to have a clear benefit. Every new addition adds complexity which has a cost both in maintainability and performance. Obscure or little used features may add conditional checks in frequently executed code paths such as read and write slowing down the overall performance of the filesystem even if they are not used.

Although we decided to come up with a new on-disk inode format, we chose not to change the format of the superblock, the cylinder group maps, or the directories. Additional information needed for the UFS2 superblock and cylinder groups is stored in spare fields of the UFS1 superblock and cylinder groups. Maintaining the same format for these data structures allows a single code base to be used for both UFS1 and UFS2. Because the only difference between the two filesystems is in the format of their inodes, code can dereference pointers to superblocks, cylinder groups, and directory entries without need of checking what type of filesystem is being accessed. To minimize conditional checking of code that references inodes, the on-disk inode is converted to a common in-core format when the inode is first read in from the disk, and converted back to its on-disk format when it is written back. The effect of this decision is that there are only nine out of several hundred routines that are specific to UFS1 versus UFS2. The benefit of having a single code base for both filesystems is that it dramatically reduces the maintenance cost. Outside of the nine filesystem format specific functions, fixing a bug in the code fixes it for both filesystem types. A common code base also means that as the symmetric multiprocessing support gets added, it only needs to be done once for the UFS family of filesystems.

Although we still use the same data structure to describe cylinder groups, the practical definition of them has changed. In the era of UFS1, the filesystem could get an accurate view of the disk geometry including the cylinder and track boundaries and could accurately compute the rotational location of every sector. Modern disks hide this information providing fictitious numbers of blocks per track, tracks per cylinder, and cylinders per disk. Indeed, in modern RAID arrays, the "disk" that is presented to the filesystem may really be composed from a collection of disks in the RAID array. While some research has been done to figure out the true geometry of a disk [Griffin et al, 2002; Lumb et al, 2002; Schindler et al, 2002], the complexity of using such information effectively is quite high. Modern disks have greater numbers of sectors per track on the outer part of the disk than the inner part which makes calculation of the rotational position of any given sector quite complex to calculate. So, for UFS2, we decided to get rid of all the rotational layout code found in UFS1 and simply assume that laying out files with numerically close block numbers (sequential being viewed as optimal) would give the best performance. Thus, the cylinder group structure is retained in UFS2, but it is used only as a convenient way to manage logically close groups of blocks. The rotational layout code had been disabled in UFS1 since the late 1980s, so as part of the code base cleanup it was removed entirely.

The UFS1 filesystem uses 32-bit inode numbers. While it is very tempting to increase these inode numbers to 64 bits in UFS2, doing so would require that the directory format be changed. There is a lot of code that works directly on directory entries. Changing directory formats would entail creating many more filesystem specific functions which would increase the complexity and maintainability issues with the code. Furthermore, the current APIs for referencing directory entries use 32-bit inode numbers. So, even if the underlying filesystem supported 64-bit inode numbers, they could not currently be made visible to user applications. In the short term, applications are not running into the four billion files-per-filesystem limit that 32-bit inode numbers impose. If we assume that the growth rate in the number of files per filesystem over the last twenty years will continue at the same rate, we estimate that the 32-bit inode number should be sufficient for another ten to twenty years. However, the limit will be reached before the 64-bit block limit of UFS2 is reached. So, the UFS2 filesystem has reserved a flag in the superblock to indicate that it is a filesystem with 64-bit inode numbers. When the time comes to begin using 64-bit

inode numbers, the flag can be turned on and the new directory format can be used. Kernels that predate the introduction of 64-bit inode numbers check this flag and will know that they cannot mount such filesystems.

Another change that was contemplated was changing to a more complex directory structure such as one that uses B-trees to speed up access for large directories. This technique is used in many modern filesystems such as XFS [Sweeney et al, 1996], JFS [Best & Kleikamp, 2003], ReiserFS [Reiser, 2001], and in later versions of Ext2 [Phillips, 2001]. We decided not to make the change at this time for several reasons. First, we had limited time and resources and we wanted to get something working and stable that could be used in the time frame of FreeBSD 5.0. By keeping the same directory format, we were able to reuse all the directory code from UFS1, did not have to change numerous filesystem utilities to understand and maintain a new directory format, and were able to produce a stable and reliable filesystem in the time frame available to us. The other reason that we felt that we could retain the existing directory structure is because of the dynamic directory hashing that was added to FreeBSD [Dowse & Malone, 2002]. The dynamic directory hashing retrofits a directory indexing system to UFS. To avoid repeated linear searches of large directories, the dynamic directory hashing builds a hash table of directory entries on the fly when the directory is first accessed. This table avoids directory scans on subsequent lookups, creates, and deletes. Unlike filesystems originally designed with large directories in mind, these indices are not saved on disk and so the system is backwards compatible. The effect of the dynamic directory hashing is that large directories in UFS cause minimal performance problems.

Borrowing the technique used by the Ext2 filesystem a flag was also added to indicate that an on-disk indexing structure is supported for directories [Phillips, 2001]. This flag is unconditionally turned off by the existing implementation of UFS. In the future, if an implementation of an on-disk directory-indexing structure is added, the implementations that support it will not turn the flag off. Index-supporting kernels will maintain the indices and leave the flag on. If an old non-index-supporting kernel is run, it will turn off the flag so that when the filesystem is once again run under a new kernel, the new kernel will discover that the indexing flag has been turned off and will know that the indices may be out date and have to be rebuilt before being used. The only constraint on an implementation of the indices is that they have to be an auxiliary data structure that references the old linear directory format.

3. Extended Attributes

A major addition in UFS2 is support for extended attributes. Extended attributes are a piece of auxiliary data storage associated with an inode that can be used to store auxiliary data that is separate from the contents of the file. The idea is similar to the concept of data forks used in the Apple filesystem [Apple, 2003]. By integrating the extended attributes into the inode itself, it is possible to provide the same integrity guarantees as are made for the contents of the file itself. Specifically, the successful completion of an "fsync" system call ensures that the file data, the extended attributes, and all names and paths leading to the names of the file are in stable store.

The current implementation has space in the inode to store up to two blocks of extended attributes. The new UFS2 inode format had room for up to five additional 64-bit pointers. Thus, the number of extended attribute blocks could have been in the range of one to five blocks. We chose to allocate two blocks to the extended attributes and to leave the other three as spares for future use. By having two, all the code had to be prepared to deal with an array of pointers, thus if the number got expanded into the remaining spares in the future the existing implementation will work without change. By saving three spares, we provided a reasonable amount of space for future needs. And, if the decision to allow only two blocks proves to be too little space, one or more of the spares can be used to expand the size of the extended attributes in the future. If vastly more extended attribute space is needed, one of the spares could be used as an indirect pointer to extended attribute data blocks.

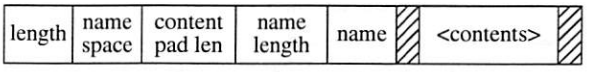

Figure 1: *Format of Extended Attributes*

Figure 1 shows the format used for the extended attributes. The header of each attribute has a 4-byte length, 1-byte name space class, 1-byte content pad length, 1-byte name length, and name. The name is padded so that the contents start on an 8-byte boundary. The contents are padded to the size shown by the "content pad length" field. Applications that do not understand the name space or name can simply skip over the unknown attribute by adding the length to their current position to get to the next attribute. Thus, many different applications can share the usage

of the extended attribute space, even if they do not understand each other's data types.

The first of two initial uses for extended attributes is to support access control lists, generally referred to as ACLs. An ACL replaces the group permissions for a file with a more specific list of the users that are permitted to access the files along with a list of the permissions that they are granted. These permissions include the traditional read, write, and execute permissions along with other properties such as the right to rename or delete the file [Rhodes, 2003].

Earlier implementations of ACLs were done with a single auxiliary file per filesystem that was indexed by the inode number and had a small and fixed sized area to store the ACL permissions. The size was small to keep the size of the auxiliary file reasonable since it had to have space for every possible inode in the filesystem. There were two problems with this implementation. The fixed size of the space per inode to store the ACL information meant that it was not possible to give access to long lists of users. The second problem was that it was difficult to atomically commit changes to the ACL list for a file since an update requires that both the file inode and the ACL file be written to have the update take effect [Watson, 2000].

Both problems with the auxiliary file implementation of ACLs are fixed by storing the ACL information directly in the extended-attribute data area of the inode. Because of the large size of the extended attribute data area (a minimum of 8 kilobytes and typically 32 kilobytes), long lists of ACL information can be easily stored. Space used to store extended attribute information is proportional to the number of inodes with extended attributes and the size of the ACL lists that they use. Atomic update of the information is much easier since writing the inode will update the inode attributes and the set of data that it references including the extended attributes in one disk operation. While it would be possible to update the old auxiliary file on every "fsync" system call done on the filesystem, the cost of doing so would be prohibitive. Here, the kernel knows if the extended attribute data block for an inode is dirty and can write just that data block during an "fsync" call on the inode.

The second use for extended attributes is for data labeling. Data labels are used to provide permissions for mandatory access controls (MACs). The kernel provides a MAC framework that permits dynamically introduced system-security modules to modify system security functionality. This framework can be used to support a variety of new security services, including traditional labeled mandatory access control models. The framework provides a series of entry points which is called by code supporting various kernel services, especially with respects to access control points and object creation. The framework then calls out to security modules to offer them the opportunity to modify security behavior at those MAC entry points. Thus, the filesystem does not codify how the labels are used or enforced. It simply stores the labels associated with the inode and produces them when a security modules needs to query them to make a permission check [Watson, 2001; Watson et al, 2003].

We considered storing symbolic links in the extended attribute area. We chose not to do this for three reasons. First, the time to access an extended storage block is the same as the time to access a regular data block. Second, since symbolic links rarely have any extended attributes, there would be no savings in storage since a filesystem fragment would be needed whether it was stored in a regular data block or in an extended storage block. Third, if it were stored in the extended storage area, it would take more time to traverse down the attribute list to find it.

4. New Filesystem Capabilities

Several other improvements were made when the enlarged inode format was created. We decided to get an early jump on the year 2038 problem (specifically, Tue Jan 19 03:14:08 2038 GMT which could be a really ugly way to usher in my 84th birthday). We expanded the time fields (which hold seconds-since-1970) for access, modification, and inode-modification times from 32-bits to 64-bits. At plus or minus 136 billion years that should carry us from well before the universe was created until long after our Sun has burned itself out. We left the nanoseconds fields for these times at 32-bits as we did not feel that added resolution was going to be useful in the foreseeable future. We considered expanding the time to only 48-bits. We chose to go to 64-bits as 64-bits is a native size that can be easily manipulated with existing and likely future architectures. Using 48-bits would have required an extra unpacking or packing step each time the field was read or written. Also, going to 64-bits ensures enough bits for all likely measured time so will not have to be enlarged.

At the same time we also added a new time field (also 64-bit) to hold the birth time (also commonly called the creation time) of the file. The birth time is set when the inode is first allocated and is not changed thereafter. It has been added to the structure

returned by the "stat" system call so that applications can determine its value and so that archiving programs such as **dump**, **tar**, and **pax** can save this value along with the other file times. The birth time was added to a previously spare field in the "stat" system call structure so that the size of the structure did not change. Thus, old versions of programs that use the "stat" call continue to work.

To date, only the **dump** program has been changed to save the birth time value. This new version of **dump** which can dump both UFS1 and UFS2 filesystems, creates a new dump format which is not readable by older versions of **restore**. The updated version of **restore** can identify and restore from both old and new dump formats. The birth times are only available and setable from the new dump format.

The "utimes" system call sets the access and modification times of a file to a specified set of values. It is used primarily by archive retrieval programs to set newly extracted files times back to those associated with the file in the archive. With the addition of birth time, we added a new system call that allows the setting of access, modification, and birth times. However, we realized that many existing applications will not be changed to use the new "utimes" system call. The result will be that the files that they retrieved from archives will have a newer birth time than access or modification times.

To provide a sensible birth time for applications that are unaware of the birth time attribute, we changed the semantics of the "utimes" system call so that if the birth time was newer than the value of the modification time that it was setting, it sets the birth time to the same time as the modification time. An application that is aware of the birth time attribute can set both the birth time and the modification time by doing two calls to "utimes". First it calls "utimes" with a modification time equal to the saved birth time, then it calls "utimes" a second time with a modification time equal to the (presumably newer) saved modification time. For filesystems that do not store birth times, the second call will overwrite the first call resulting in the same values for access and modification times as they would have previously gotten. For filesystems that support birth time, it will be properly set. And most happily for the application writers, they will not have to conditionally compile the name of "utimes" for BSD and non-BSD systems. They just write their applications to call the standard interface twice knowing that the right thing will happen on all systems and filesystems. For those applications that value speed of execution over portability can use the new version of the "utimes" system call that allows all time values to be set with one call.

Another incremental change to the inode format was to split the flags field into two separate 32-bit fields, one for flags that can be set by applications (as in UFS1) and a new field for flags maintained strictly by the kernel. An example of a kernel flag is the SNAPSHOT flag used to label a file as being a snapshot. Another kernel-only flag is OPAQUE which is used by the union filesystem to mark a directory which should not make the layers below it visible. By moving these kernel flags into a separate field, they will not be accidentally set or cleared by a naive or malicious application.

4.1. Dynamic Inodes

One of the common complaints about the UFS1 filesystem is that it preallocates all its inodes at the time that the filesystem is created. For filesystems with millions of files, the initialization of the filesystem can take several hours. Additionally, the filesystem creation program, **newfs**, had to assume that every filesystem would be filled with many small files and allocate a lot more inodes than were likely to ever be used. If a UFS1 filesystem uses up all its inodes, the only way to get more is to dump, rebuild, and restore the filesystem. The UFS2 filesystem resolves these problems by dynamically allocating its inodes. The usual implementation of dynamically allocated inodes requires a separate filesystem data structure (typically referred to as the inode file) that tracks the current set of inodes. The management and maintenance of this extra data structure adds overhead and complexity and often degrades performance.

To avoid these costs, UFS2 preallocates a range of inode numbers and a set of blocks for each cylinder group. Initially each cylinder group has a single block of inodes allocated (a typical block holds 32 or 64 inodes). When the block fills up, the next block of inodes in the set is allocated and initialized. The set of blocks that may be allocated to inodes is held as part of the free-space reserve until all other space in the filesystem is allocated. Only then can it be used for file data.

In theory a filesystem could fill using up all the blocks set aside for inodes. Later after large files had been removed and many small files created to replace them, the filesystem might find itself unable to allocated the needed inodes because all the space set aside for inodes was still in use. Here, it would be necessary to reallocate existing files to move them to new locations outside of the inode area. Such code has not been written as we do not anticipate that this

condition will arise in practice as the free space reserve used on most filesystems (8%) exceeds the amount of space needed for inodes (typically 2-6%). On these systems only a process running with root privileges would ever be able to allocate the inode blocks. Should the code prove necessary in actual use, it can be written at that time. Until it is written, filesystems hitting this condition will return an "out of inodes" error on attempts to create new files.

One of the side benefits of dynamically allocating inodes is that the time to create a new filesystem in UFS2 is about 1 percent of the time that it takes in UFS1. A filesystem that would take one hour to build in a UFS1 format can be built in under a minute in the UFS2 format. While filesystem creations are not a common operation, having them build quickly does matter to the system administrators that have to do such tasks with some regularity.

The cost of dynamically allocating inodes is one extra disk write for every 64 new inodes that are created. Although this cost is quite low compared to the other costs of creating 64 new files, some systems administrators might want to preallocate more than the minimal number of inodes. If such a demand arises, it would be trivial to add a flag to the **newfs** program to preallocate additional inodes at the time that the filesystem is created.

4.2. Boot Blocks

The UFS1 filesystem reserved an 8 kilobyte space at the beginning of the filesystem in which to put a boot block. While this space seemed huge compared to the 1 kilobyte book block that it replaced, over time it has gotten increasingly difficult to cram the needed boot code into this space. Consequently we decided to revisit the boot block size in UFS2.

The boot code has a list of locations to check for boot blocks. A boot block can be defined to start at any 8 kilobyte boundary. We set up an initial list with four possible boot block sizes: none, 8 kilobytes, 64 kilobytes, and 256 kilobytes. Each of these locations was selected for a particular purpose. Filesystems other than the root filesystem do not need to be bootable, so can use a boot block size of zero. Also, filesystems on tiny media that need every block that they can get such as floppy disks can use a zero size boot block. For architectures with simple boot blocks, the traditional UFS1 8 kilobyte boot block can be used. More typically the 64 kilobyte boot block is used (for example on the PC architecture with its need to support booting from a myriad of busses and disk drivers).

We added the 256 kilobyte boot block in case some architecture or application needs to set aside a particularly large boot area. While this was not strictly necessary as new sizes can be added to the list at any time, it can take a long time before the updated list gets propagated to all the boot programs and loaders out on the existing systems. By adding the option for a huge boot area now, we can ensure it will be readily available should it be needed on short notice in the future.

One of the unexpected side effects of using a 64 kilobyte boot block for UFS2 is that if the partition had previously had a UFS1 filesystem on it, the superblock for the former UFS1 filesystem may not be overwritten. If an old version of **fsck** that does not first look for a UFS2 filesystem is run and finds the UFS1 superblock, it can incorrectly try to rebuild the UFS1 filesystem destroying the UFS2 filesystem in the process. So, when building UFS2 filesystems, the **newfs** utility looks for old UFS1 superblocks and zeros them out.

5. Changes and Enhancements to Soft Updates

Traditionally, filesystem consistency has been maintained across system failures either by using synchronous writes to sequence dependent metadata updates or by using write-ahead logging to atomically group them [Seltzer et al, 2000]. Soft updates, an alternative to these approaches, is an implementation mechanism that tracks and enforces metadata update dependencies to ensure that the disk image is always kept consistent. The use of soft updates obviates the need for a separate log or for most synchronous writes [McKusick & Ganger, 1999].

The addition of extended attribute data to the inode required that the soft updates code be extended so that it could ensure the integrity of these new data blocks. As with the file data blocks, it ensures that the extended data blocks and the bitmaps that show that they are in use are written to disk before they are claimed by the inode. Soft updates also ensure that any updated extended attribute data is committed to disk as part of an "fsync" of the file.

Two important enhancements were made to the existing soft updates implementation. These enhancements were initially made for UFS2 but because of the shared code base with UFS1 were trivially integrated to work with UFS1 filesystems as well.

When a file is removed on a filesystem running with soft updates, the removal appears to happen very quickly, but the process of removing the file and

returning its blocks to the free list may take up to several minutes. Prior to UFS2, the space held by the file did not show up in the filesystem statistics until the removal of the file had been completed. Thus, applications that clean up disk space such as the news expiration program would often vastly overshoot their goal. They work by removing files and then checking to see if enough free space has showed up. Because of the time lag in having the free space recorded, they would remove far too many files. To resolve problems of this sort, the soft updates code now maintains a counter that keeps track of the amount of space that is held by the files that it is in the process of removing. This counter of pending space is added to the actual amount of free space as reported by the kernel (and thus by utilities like **df**). The result of this change is that free space appears immediately after the "unlink" system call returns or the **rm** utility finishes.

The second and related change to soft updates has to do with avoiding false out-of-space errors. When running with soft updates on a nearly full filesystem with high turnover rate (for example when installing a whole new set of binaries on a root partition), the filesystem can return a filesystem full error even though it reports that it has plenty of free space. The filesystem full message happens because soft updates has not managed to free the space from the old binaries in time for it to be available for the new binaries.

The initial attempt to correct this problem was to simply have the process that wished to allocate space wait for the free space to show up. The problem with this approach is that it often had to wait for up to a minute. In addition to making the application seem intolerably slow, it usually held a locked vnode which could cause other applications to get blocked waiting for it to become available (often referred to as a lock race to the root of the filesystem). Although the condition would clear in a minute or two, users often assumed that their system had hung and would reboot.

To remedy this problem, the solution devised for UFS2 is to co-opt the process that would otherwise be blocked and put it to work helping soft updates process the files to be freed. The more processes trying to allocate space, the more help that is available to soft updates and the faster free blocks begin to appear. Usually in under one second enough space shows up that the processes can return to their original task and proceed to completion. The effect of this change is that soft updates can now be used on small nearly full filesystems with high turnover.

6. Enhancements for Live Dumps

A filesystem snapshot is a frozen image of a filesystem at a given instant in time. Snapshots support several important features: the ability to provide back-ups of the filesystem at several times during the day, the ability to do reliable dumps of live filesystems, and the ability to run a filesystem check program on a active system to reclaim lost blocks and inodes [McKusick, 2002].

With the advent of filesystem snapshots, the **dump** program has been enhanced to safely dump live filesystems. When given the -L flag, **dump** verifies that it is being asked to dump a mounted filesystem, then takes a snapshot of the filesystem and dumps the snapshot instead of on the live filesystem. When **dump** completes, it releases the snapshot.

The initial implementation of live dumps had the **dump** program do the "mount" system call itself to take the snapshot. However, most systems require root privilege to use the "mount" system call. Since dumps are often done by the *operator* user rather than *root*, an attempt to take a snapshot will fail.

To get around this problem, a new set-user-identifier *root* program was written called **mksnap_ffs**. The **mksnap_ffs** command creates a snapshot with a given name on a specified filesystem. The snapshot file must be contained within the filesystem being snapshotted. The group ownership of the file is set to *operator*; the owner of the file remains *root*. The mode of the snapshot is set to be readable by the owner or members of the *operator* group.

The **dump** program now invokes **mksnap_ffs** to create the snapshot rather than trying to create it directly. The result is that anyone with *operator* privileges can now reliably take live dumps. Allowing *operator* group access to the snapshot does not open any new security holes since the raw disk is also readable by members of the *operator* group (for the benefit of traditional **dump**). Thus, the information that is available in the snapshot can also be accessed directly through the disk device.

7. Large Filesystem Snapshots

Creating and using a snapshot requires random access to the snapshot file. The creation of a snapshot requires the inspection and copying of all the cylinder group maps. Once in operation, every write operation to the filesystem must check whether the block being written needs to be copied. The information on whether a blocks needs to be copied is contained in the snapshot file metadata (its indirect blocks).

Ideally, this metadata would be resident in the kernel memory throughout the lifetime of the snapshot. In FreeBSD, the entire physical memory on the machine can be used to cache file data pages if the memory is not needed for other purposes. Unfortunately, data pages associated with disks can only be cached in pages mapped into the kernel physical memory. Only about 10 megabytes of kernel memory is dedicated to such purposes. Assuming that we allow up to half of this space to be used for any single snapshot, the largest snapshot whose metadata that we can hold in memory is 11 megabytes. Without help, such a tiny cache would be hopeless in trying to support a multi-terabyte snapshot.

In an effort to support multi-terabyte snapshots with the tiny metadata cache available, it is necessary to observe the access patterns on typical filesystems. The snapshot is only consulted for files that are being written. The filesystem is organized around cylinder groups which maps small contiguous areas of the disk. Within a directory, the filesystem tries to allocate all the inodes and files in the same cylinder group. When moving between directories different cylinder groups are usually inspected. Thus, the widely random behavior occurs from movement between cylinder groups. Once file writing activity settles down into a cylinder group, only a small amount of snapshot metadata needs to be consulted. That metadata will easily fit in even the tiny kernel metadata cache. So, the need is to find a way to avoid thrashing the cache when moving between cylinder groups.

The technique used to avoid thrashing when moving between cylinder groups is to build a look aside table of all the blocks that were copied during the time that the snapshot was made. This table lists the blocks associated with all the snapshot metadata blocks, the cylinder groups maps, the super block, and blocks that contain active inodes. When a copy-on-write fault occurs for a block, the first step is to consult this table. If the block is found in the table, then no further searching needs to be done in any of the snapshots. If the block is not found, then the metadata of each active snapshot on the filesystem must be consulted to see if a copy is needed. This table lookup saves time as it not only avoids faulting in metadata for widely scattered blocks, but it also avoids the need to consult potentially many snapshots.

Another problem with snapshots on large filesystems is that they aggravated existing deadlock problems. When there are multiple snapshots associated with a filesystem, they are kept in a list ordered from oldest to youngest. When a copy-on-write fault occurs, the list is traversed letting each snapshot decide if it needs to make a copy of the block that is about to be written. Originally, each snapshot inode had its own lock. A deadlock could occur between two processes each trying to do a write. Consider the example in Fig. 2. It shows a filesystem with two snapshots, snap1 and snap2. Process A holds snapshot 1 locked and process B holds snapshot 2 locked. Both snap1 and snap2 have decided that they need to allocate a new block in which to hold a copy of the block being written by the process that holds them locked. The writing of the new block in snapshot 1 will cause the kernel running in the context of process A to scan the list of snapshots which will get blocked at snapshot 2 because it is held locked by process B. Meanwhile, the writing of the new block in snapshot 2 will cause the kernel running in the context of process B to scan the list of snapshots which will get blocked at snapshot 1 because it is held locked by process A.

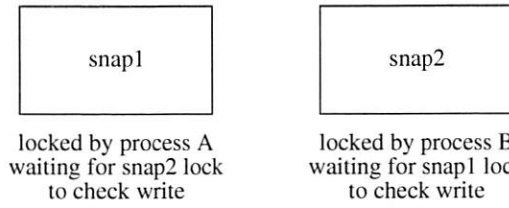

Figure 2: *Snapshot deadlock scenario*

The resolution to the deadlock problem is to allocate a single lock that is used for all the snapshots on a filesystem. When a new snapshot is created, the kernel checks whether there are any other snapshots on the filesystem. If there are, the per-file lock associated with the new snapshot inode is released and replaced with the lock used for the other snapshots. With only a single lock, the access to the snapshots as a whole are serialized. Thus, in Fig. 2, process B will hold the lock for all the snapshots and will be able to make the necessary checks and updates while process A will be held waiting. Once process B completes its scan, process A will be able to get access to all the snapshots and will be able to run successfully to completion. Because of the added serialization of the snapshot lookups, the look-aside table described earlier is important to ensure reasonable performance of snapshots. In gathering statistics on our running systems, we found that the look-aside table resolves nearly half of the snapshot copy-on-write lookups. Thus, we found that the look-aside table keeps the contention for the snapshot lock to a reasonable level.

8. Running Fsck on Large Filesystems

Traditionally, after an unclean system shutdown, the filesystem check program, **fsck**, has had to be run over all inodes in a filesystem to ascertain which inodes and blocks are in use and to correct the bitmaps. The current implementation of soft updates guarantees the consistency of all filesystem resources, including the inode and block bitmaps. With soft updates, the only inconsistency that can arise in the filesystem (barring software bugs and media failures) is that some unreferenced blocks may not appear in the bitmaps and some inodes may have to have overly high link counts reduced. Thus, it is completely safe to begin using the filesystem after a crash without first running **fsck**. However, some filesystem space may be lost after each crash. Thus, there is a version of **fsck** that can run in the background on an active filesystem to find and recover any lost blocks and adjust inodes with overly high link counts. A special case of the overly high link count is one that should be zero. Such an inode will be freed as part of reducing its link count to zero. This garbage collection task is less difficult than it might at first appear, since this version of **fsck** only needs to identify resources that are not in use and cannot be allocated or accessed by the running system [McKusick & Ganger, 1999].

With the addition of snapshots, the task becomes simple, requiring only minor modifications to the standard **fsck**. When run in background cleanup mode, **fsck** starts by taking a snapshot of the filesystem to be checked. **Fsck** then runs over the snapshot filesystem image doing its usual calculations just as in its normal operation. The only other change comes at the end of its run, when it wants to write out the updated versions of the bitmaps. Here, the modified **fsck** takes the set of blocks that it finds were in use at the time of the snapshot and removes this set from the set marked as in use at the time of the snapshot—the difference is the set of lost blocks. It also constructs the list of inodes whose counts need to be adjusted. **Fsck** then uses a new system call to notify the filesystem of the identified lost blocks so that it can replace them in its bitmaps. It also gives the set of inodes whose link counts need to be adjusted; those inodes whose link count is reduced to zero are truncated to zero length and freed. When **fsck** completes, it releases its snapshot [McKusick, 2002].

As filesystems have gotten bigger the time to run either a foreground or a background **fsck** has increased to multiple hours. Being able to run **fsck** in background has largely mitigated the running time issue because it allows normal system operation to proceed in parallel.

Another problem with running **fsck** on large filesystems is that the memory that it consumes grows in proportional to the size of the filesystem being checked. The main consumption of memory is four bytes per regular inode, 40 to 50 bytes per directory inode, and one bit per filesystem data block. On a typical UFS2 filesystem with 16 kilobyte blocks and 2 kilobyte fragments, the data-block map requires 64 megabytes of memory per terabyte of filesystem. Because UFS2 does not preallocate inodes, but rather allocates the inodes as they are needed, the memory required is dependent on the number of files that are created in the filesystem.

Filesystem	Files per Tb	Dirs per Tb	**fsck** memory per Tb	maximum checkable filesystem
/usr	93M	15M	1200K	3Tb
/jukebox	243K	18K	66K	60Tb

Table 1: *Maximum filesystem sizes checkable by* **fsck** *on a 32-bit architecture*

The number of files and directories in a filesystem make a huge difference in the amount of memory required by **fsck**. Table 1 shows the two ends of the spectrum. At one end is a typical FreeBSD **/usr** filesystem assuming that it grew at its current file and directory mix to fill a 1 terabyte disk. The memory footprint of **fsck** is dominated by the memory to manage the inodes and would require the entire address space of a 32-bit processor for a filesystem of about 3 terabytes. At the other extreme is the author's **/jukebox** filesystem assuming that it grew at its current file and directory mix to fill a 1 terabyte disk. The memory footprint of **fsck** is dominated by the memory to manage the data blocks and would require the entire address space of a 32-bit processor for a filesystem of about 60 terabytes. My expectation is that as disks get larger, they will tend to be filled with larger files of audio and video. Thus, in practice **fsck** will run out of space on 32-bit architectures at about 30 terabyte filesystems. Hopefully by the time that such filesystems are common, they will be running on 64-bit architectures.

In the event that **fsck** hits the memory limit of 32-bit architectures, Julian Elischer has suggested that one solution is to implement an "offline, non-in-place" version of **fsck** using all those techniques we learned in CS101 relating to mag-tape merge sorts. **Fsck** would have to have a small (20 gigabyte) disk partition set aside to hold working files, to which it would write files of records detailing block numbers,

etc. Then it would do merge or block sorts on those files to get them in various orders (depending on fields in the records). **Fsck** would then recombine them to find such things as multiple referenced blocks and other file inconsistencies. It would be slow, but at least it could be used to check a 100 terabyte array, where the in-memory version would need a process VM space of 13 Gigabytes which is clearly impossible on the 32-bit PC.

Journalling filesystems provide a much faster state recovery than **fsck**. For this reason, there is ongoing work to provide a journalling option for UFS2. However, even journalling filesystems need to have a filesystem recovery program such as **fsck**. In the event of media or software failure, the filesystem can be damaged in ways that the journal cannot fix. Thus, the size of the recovery program is an issue for all filesystems. Indeed, the fact that UFS needs to use **fsck** in its general operation ensures that **fsck** is kept in good working order and is known to work even on very large filesystems.

9. Performance

The performance of UFS2 is nearly identical to that of UFS1. This similarity in performance is hardly surprising since the two filesystem share most of the same code base and use the same allocation algorithms. The purpose of UFS2 was not to try and improve on the performance of UFS1 which is already within 80-95% of the bandwidth of the disk. Rather it was to support multi-terabyte filesystems and to provide new capabilities such as extended attributes without losing performance. It has been successful in that goal.

10. Future Work

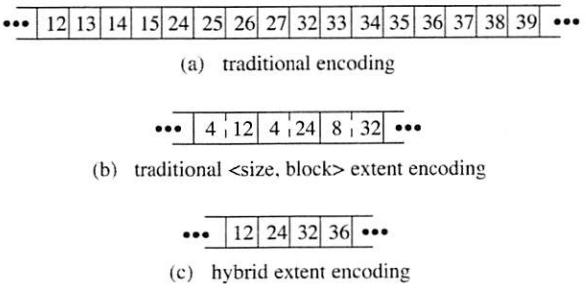

Figure 3: *Alternative file metadata representations*

With the addition of dynamic block reallocation in the early 1990s [Seltzer & Smith, 1996], the UFS1 filesystem has had the ability to allocate most files contiguously on the disk. The metadata describing a large file consists of indirect blocks with long runs of sequential block numbers, see Fig. 3-(a). For quick access while a file is active, the kernel tries to keep all of a file's metadata in memory. With UFS2 the space required to hold the metadata for a file is doubled as every block pointer grows from 32-bits to 64-bits. To provide a more compact representation, many filesystems use an extent-based representation. A typical extent-based representation uses pairs of block numbers and lengths. Figure 3-(b) represents the same set of block number as Fig. 3-(a) in an extent-based format. Provided that the file can be laid out nearly contiguously, this representation provides a very compact description. However, randomly or slowly written files can end up with many non-contiguous block allocations which will produce a representation that requires more space than the one used by UFS1. This representation also has the drawback that it can require a lot of computation to do random access to the file since the block number needs to be computed by adding up the sizes starting from the beginning of the file until the desired seek offset is reached.

To gain most of the efficiencies of extends without the random access inefficiencies, UFS2 has added a field to the inode that will allow that inode to use a larger block size. Small, slowly growing, or sparse files set this value to the regular filesystem block size and represent their data in the traditional way show in Fig. 3-(a). However, when the filesystem detects a large dense file, it can set this inode-block-size field to a value two to sixteen times the filesystem block size. Figure 3-(c) represents the same set of block number as Fig. 3-(a) with the inode-block-size field set to four times the filesystem block size. Each block pointer references a piece of disk storage that is four times larger which reduces the metadata storage requirement by 75 percent. Since every block pointer other than possibly the last one references an equal sized block, computation of random access offsets is just as fast as in the traditional metadata representation. It also cannot degrade to a larger representation than the traditional metadata representation.

The drawback to this approach is that once a file has committed to using a larger block size, it can only utilize blocks of that size. If the filesystem runs out of big blocks then the file can no longer grow and either the application will get an "out-of-space" error, or the filesystem has to recreate the metadata with the standard filesystem block size. My current plan is to write the code to recreate the metadata. While recreating the metadata usually will cause a long pause, We expect that condition to be quite rare and not a

noticeable problem in actual use.

11. Current Status

The UFS2 filesystem was developed for the FreeBSD Project by the author under contract to Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program. Under the terms of that contract, the software must be released under a Berkeley-style copyright. The UFS2 filesystem was written in 2002 and first released in FreeBSD 5.0. Extensive user feedback in that release has been helpful in shaking out latent short-comings particularly in the ability of UFS2 to smoothly handle the really big filesystems for which it was designed. The biggest current limitation is that the disk labels used in FreeBSD 5.0 can only describe 2 terabyte disks. We are hoping that the new larger disk labels will be available by the time FreeBSD 5.1 is released.

12. References

Apple, 2003.

Apple, "Mac OS X Essentials, Chapter 9 Filesystem, Section 12 Resource Forks," *http://developer.apple.com/techpubs/macosx/Essentials/SystemOverview/FileSystem/chapter_9_section_12.html* (2003).

Best & Kleikamp, 2003.

S. Best & D. Kleikamp, "How the Journaled File System handles the on-disk layout," *http://www-106.ibm.com/developerworks/linux/library/l-jfslayout/* (2003).

Dowse & Malone, 2002.

I. Dowse & D. Malone, "Recent Filesystem Optimizations on FreeBSD," *Proceedings of the Freenix Track at the 2002 Usenix Annual Technical Conference,* p. 245–258 (June 2002).

Griffin et al, 2002.

J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, & G. R. Ganger, "Timing-accurate Storage Emulation," *Proceedings of the Usenix Conference on File and Storage Technologies,* p. 75–88 (January 2002).

Lumb et al, 2002.

C. R. Lumb, J. Schindler, & G. R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," *Proceedings of the Usenix Conference on File and Storage Technologies,* p. 275–288 (January 2002).

McKusick, 2002.

M. McKusick, "Running Fsck in the Background," *Proceedings of the BSDCon 2002 Conference,* p. 55–64 (February 2002).

McKusick et al, 1996.

M. McKusick, K. Bostic, M. Karels, & J. Quarterman,, *The Design and Implementation of the 4.4BSD Operating System,* p. 269–271. Addison Wesley Publishing Company, Reading, MA (1996).

McKusick & Ganger, 1999.

M. McKusick & G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of the Freenix Track at the 1999 Usenix Annual Technical Conference,* p. 1–17 (June 1999).

McKusick et al, 1984.

M. McKusick, W. Joy, S. Leffler, & R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* 2, 3, p. 181–197 (August 1984).

Phillips, 2001.

D. Phillips, "A Directory Index for Ext2," *Proceedings of the Usenix Fifth Annual Linux Showcase and Conference* (November 2001).

Reiser, 2001.

H. Reiser, "The Reiser File System," *http://www.namesys.com/res_whol.shtml* (January 2001).

Rhodes, 2003.

T. Rhodes, "FreeBSD Handbook, Chapter 3, Section 3.3 File System Access Control Lists," *http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/fs-acl.html* (2003).

Schindler et al, 2002.

J. Schindler, J. L. Griffin, C. R. Lumb, & G. R. Ganger, "Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics," *Proceedings of the Usenix Conference on File and Storage Technologies,* p. 259–274 (January 2002).

Seltzer et al, 2000.

M. Seltzer, G. Ganger, M. McKusick, K. Smith, C. Soules, & C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," *Proceedings of the San Diego Usenix Conference,* p. 71–84 (June 2000).

Seltzer & Smith, 1996.

M. Seltzer & K. Smith, "A Comparison of FFS Disk Allocation Algorithms," *Winter USENIX Conference,* p. 15–25 (January 1996).

Sweeney et al, 1996.

A. Sweeney, D. Doucette, C. Anderson, W. Hu, M. Nishimoto, & G. Peck, "Scalability in the XFS File System," *Proceedings of the 1996 Usenix Annual Technical Conference*, p. 1–14 (January 1996).

Watson, 2000.

R. Watson, "Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD," *Proceedings of the BSDCon 2000 Conference* (September 2000).

Watson, 2001.

R. Watson, "TrustedBSD: Adding Trusted Operating System Features to FreeBSD," *Proceedings of the Freenix Track at the 2001 Usenix Annual Technical Conference* (June 2001).

Watson et al, 2003.

R. Watson, W. Morrison, C. Vance, & B. Feldman, "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0," *Proceedings of the Freenix Track at the 2003 Usenix Annual Technical Conference* (June 2003).

13. Biography

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem. One day, he hopes to see them merged seamlessly. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a doctoral degree in Computer Science. He is president of the Usenix Association, and is a member of ACM and IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at http://www.mckusick.com/~mckusick/) in the basement of the house that he shares with Eric Allman, his domestic partner of 24-and-some-odd years. You can contact him via email at <mckusick@mckusick.com>.

Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions

Hideki Eiraku
College of Information Sciences, University of Tsukuba
hdk@coins.tsukuba.ac.jp, http://www.coins.tsukuba.ac.jp/~hdk/

Yasushi Shinjo
Institute of Information Sciences and Electronics, University of Tsukuba
yas@is.tsukuba.ac.jp, http://www.is.tsukuba.ac.jp/~yas/

Abstract

A user-level operating system (OS) can be implemented as a regular user process on top of another host operating system. Conventional user-level OSes, such as User Mode Linux, view the underlying host operating system as a specific hardware architecture. Therefore, the implementation of a user-level OS often requires porting of an existing kernel to a new hardware architecture. This paper proposes a new implementation method of user-level OSes by using partial emulation of hardware and static rewriting of machine instructions. In this method, privileged instructions and their related non-privileged instructions in a native operating system are statically translated into subroutine calls that perform emulation. The translated instructions of the user-level OS are executed by both the real CPU and the partial emulator. This method does not require detailed knowledge about kernel internals. By using the proposed method, NetBSD and FreeBSD are executed as user processes on NetBSD and Linux.

1 Introduction

Running multiple operating systems (OSes) simultaneously over a single hardware platform has recently become a popular system structuring approach that offers a number of benefits [SVL01] [Pap00] [Dik00]. First, application programs written for different operating systems, such as Unix and Windows, can be simultaneously executed on a single computer. Second, several versions of a single operating system, such as MacOS9 and MacOSX, can co-exist on the same platform. Other benefits include virtual hosting and easier system management and maintenance [HH79] [SVL01] [Pap00] [Dik00].

There are two prominent approaches to running multiple operating systems over a single hardware platform: Virtual machines [LDG+03] [Law03] and user-level operating systems [Dik00] [AAS94] [Tad92]. Virtual machines provide isolated execution environments for multiple operating system kernels, which can run over the native hardware. A user-level operating system is an operating system that runs as a regular user process on another host operating system. Conventional user-level OSes view the underlying host operating system as a specific hardware architecture. Therefore, the implementation of a user-level OS often requires porting of an existing kernel to a new hardware architecture. For example, User Mode Linux (UML) [Dik00], which is a user-level OS that runs on Linux and provides another Linux system image, adds a new architecture called *um* based on the i386 architecture. In general, such porting involves significant implementation effort, and requires detailed knowledge about the kernel and the base and new architectures. In porting of User Mode Linux, the size of the new *um*-dependent part is 33,000 lines while the size of the base i386-dependent part is 40,000 lines.

In this paper, we propose a new implementation method of user-level OSes with partial emulation of hardware and rewriting of machine instructions at compile time. The key idea is to enable the execution of most instructions by the real CPU with the exception of privileged instructions, hardware interrupts, and the interaction with some peripheral

devices, which are emulated. We call this type of emulator a *partial emulator* or a *lightweight virtual machine* (LVM) because such a program does not have to emulate typical instructions, such as load, store, and arithmetic operations. In contrast, we refer to an emulator that executes all instructions as a *full emulator*.

In our implementation method, we emulate all privileged instructions. In addition, we emulate some non-privileged instructions that are tightly related with the privileged instructions. It is easy to detect execution of privileged instructions because the real CPU throws privilege violation exceptions. However, it is not trivial to detect execution of such non-privileged instructions.

To solve this problem, we use static rewriting of machine instructions at compile-time in two ways. One way is to insert an illegal instruction before each non-privileged instruction to be detected. Another way is to replace privileged instructions and related non-privileged instructions with subroutine calls for emulation. The translated instructions of a user-level OS are executed by both the real CPU and the partial emulator. By using our proposed method, we can generate a user-level OS based on a native OS without detailed knowledge about user-level OS internals. Furthermore, we can catch up the evolution of the base native OS easily. One disadvantage of our method is that we require source code of the user-level OS.

By using the proposed method, NetBSD and FreeBSD kernels are executed as user processes on NetBSD and Linux. Our user-level NetBSD on Linux is faster than NetBSD on Bochs [LDG+03], a full PC emulator, by a factor of 10. However, our user-level NetBSD is slower than NetBSD on VMware and User Mode Linux on Linux. From the experiments results, we show that the main sources of slowdown are the emulation of memory mapping hardware and the redirections of system calls and page faults.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 describes the emulation of privileged and non-privileged instructions, the redirections of system calls and page faults, and the emulation of memory mapping hardware. Section 4 shows modifications of the NetBSD kernel for hosting our partial emulator. Section 5 describes modifications of the NetBSD kernel and the FreeBSD kernel for running as user processes. Section 6 shows the performance of the user-level NetBSD. Section 7 shows future directions, and Section 8 concludes the paper.

2 Related work

Running OSes as user-level processes has been proposed in the context of microkernel system research. For example, the Mach microkernel hosts BSDs, Linux, Hurd, and other systems [GDFR90]. In a microkernel-based system, the kernel provides primitive interprocess communication, memory management, and CPU scheduling. The OS servers outside the kernel implement file systems, network protocols, etc.

It is much easier to implement an OS server on a microkernel than to implement a monolithic kernel for bare hardware. However, in the case when we already have a native kernel for bare hardware, we have to port the native kernel to the microkernel. This porting sometimes involves significant effort. A native kernel accesses hardware directly and uses interrupts and privileged instructions. Accessing hardware should be replaced with using microkernel's facilities. In this paper, we show a method that translates a native kernel for bare hardware into a user-level OS with less effort.

The idea of nesting operating systems or virtual machines had appeared even in early virtual machines for mainframes [HH79] [LW73]. Apertos is a modern object-oriented operating system that supports nesting of operating systems or *meta-objects*[Yok92]. Fluke is another modern operating system [FHL+96]. Fluke also supports efficient nesting or *recursion* with a microkernel technology. Both Apertos and Fluke have been designed to support nesting from scratch. Our method deals with commodity operating systems that are designed to run on bare hardware.

Instructional operating systems are often designed as user-level operating systems. SunOS Minix [AAS94] and VXinu [Tad92] have different structures from their native systems, PC Minix and PDP-11 Xinu, respectively. In our method, a user-level operating system has the same structure as the corresponding native operating system.

Plex86 is a virtual machine for Pentium [Law03].

Plex86 uses a special protection mechanism of Pentium, which is also known as Protected Mode Virtual Interrupts (PVI). To use this mechanism, Plex86 needs a kernel module. Plex86 provides hardware access, such as disks and networks, via a Hardware Abstraction Layer (HAL). Compared with Plex86, our approach differs in that we use a language processor (the assembler preprocessor), and we rewrite machine instructions of a user-level OS statically.

Some BSD kernels [HMM03] [LF03] have the facility to emulate other operating systems, such as Linux. In such environments, application programs written for different operating systems can be simultaneously executed on a single computer. Virtual machines and user-level OSes, including our approach, allow executing not only application programs but also operating system kernels.

Rewriting of machine instructions is used for address sandboxing with software [WLAG93]. To enforce a module to access a range of memory, this method inserts some masking instructions before each load or store instruction. In this paper, we use rewriting of machine instructions for realizing user-level OSes.

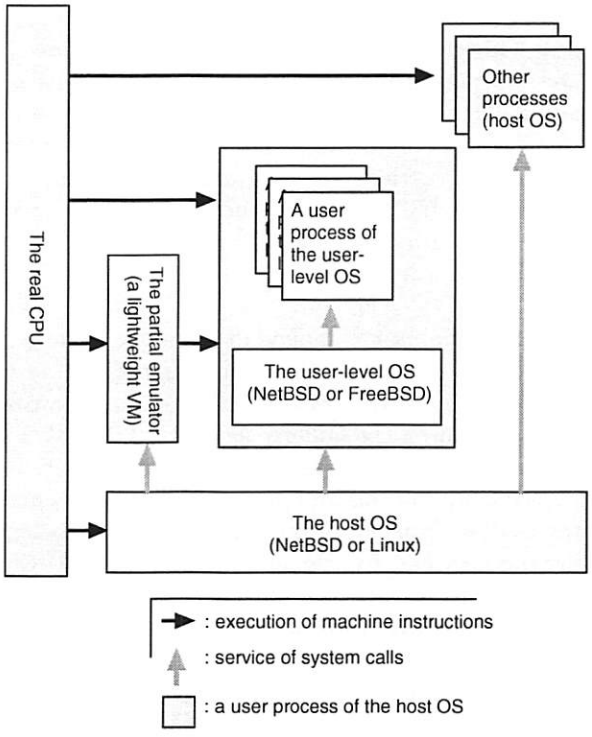

Figure 1: Running an user-level operating system and its user process by the real CPU and a partial emulator.

3 Running a user-level OS by partial emulation and rewriting of machine instructions

In this section, we propose a new implementation method of a user-level OS by using partial emulation of hardware and static rewriting of machine instructions. In this method, we have to solve the following problems:

- Detect and emulate privileged instructions and some non-privileged instructions.
- Redirect system calls and page faults to the user-level OS.
- Emulate Memory Management Unit (MMU).
- Emulate essential peripheral devices.

3.1 Detecting execution of privileged instructions and some non-privileged instructions

Although an operating system kernel includes privileged instructions, most parts are built from non-privileged instructions. If we can prepare an appropriate address space for the kernel, we can execute the most parts of the kernel by the real CPU, directly. The rest of the tasks are emulation of hardware and execution of privileged instructions. In this case, we have to emulate only a small number of CPU instructions and peripheral devices because most CPU instructions are executed by the real CPU. We call this type of emulator *a partial emulator* or *a lightweight virtual machine.*

Figure 1 shows execution of a user-level OS by the real CPU, the host OS, and the partial emulator. The user-level OS and its own user processes are included in a regular process of the host OS. Regular processes of the host OS are served with the real CPU and the host OS. The real CPU executes their machine instructions, and the host OS handles

system calls. In addition to the real CPU and the host OS, the user-level OS and its user processes are interpreted by the partial emulator. This partial emulator emulates privileged instructions and some peripheral devices, but it does not emulate normal instructions, such as arithmetic operations, load, store, and branch instructions. This is in contrast to *full emulators*, such as Bochs, which execute all CPU instructions.

A full emulator is independent of the underlying CPU, so it can execute machine instructions of another CPU. On the other hand, a partial emulator is dependent on the underlying CPU to directly execute machine instructions. Within this limitation, a partial emulator has an advantage over a full emulator in that application programs with no privileged instructions can run as fast as the real CPU. Furthermore, it is much easier to implement a partial emulator than to implement a full emulator.

The real problem on emulation is to detect the executions of some *non-privileged* instructions that are tightly coupled with corresponding privileged instructions. For example, `ltr` (load task register) of IA-32 is a privileged instruction while `str` (store task register) is a non-privileged instruction. We have to detect the executions of both `ltr` and `str`.

To detect execution of such non-privileged instructions, we use the following two methods:

insertion: Insert an illegal instruction statically before every non-privileged instruction to be detected.

rewriting: Rewrite the non-privileged and privileged instructions with subroutine calls that emulate these instructions.

3.1.1 Inserting an illegal instruction

In the insertion method, we insert an illegal instruction for each non-privileged instruction to be detected. The following is an example of insertion:

```
Before:
        str     %eax
After:
        .byte   0x8e, 0xc8
        str     %eax
```

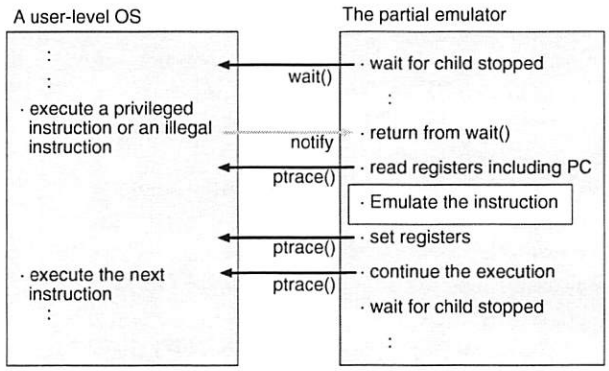

Figure 2: Execution of a privileged instruction or an illegal instruction by the partial emulator in the insertion method.

Since the byte sequence "`0x8e, 0xc8`" works as an illegal instruction in IA-32, we can detect the execution of the `str` instruction in the parent process of the user-level OS through the signal facility and the process trace facility. The parent process is the partial emulator.

Figure 2 shows execution of a privileged instruction or an inserted illegal instruction by the partial emulator. The partial emulator is a parent process of the user-level OS, and it is usually waiting for the child process being stopped with the `wait()` system call. When the user-level OS executes a privileged instruction or an inserted illegal instruction, the partial emulator is notified as if the child process received a signal (`SIGILL`). Next, the partial emulator reads the registers including the program counter with the `ptrace()` system call, and fetches the instruction pointed by the program counter. If the instruction is a privileged instruction, it is handled by the partial emulator. If the instruction is an inserted illegal instruction, the partial emulator fetches and handles the next instruction. After that, the partial emulator adjust the program counter by setting the registers with the `ptrace()` system call. Finally, the partial emulator continues the execution with the `ptrace()` system call, and is going to wait for the child process being stopped again. In summary, the partial emulator has to issue four system calls for each execution of a privileged or non-privileged instruction to be detected.

We have implemented a partial emulator for IA-32 based on the proposed method. The partial emulator consists of 1,500 lines of C code and 30 lines of assembly language code. Our partial emulator

is much smaller than Bochs that consists of 50,000 lines of C++ code.

We have also implemented an assembler preprocessor for rewriting machine instructions. For IA-32, this preprocessor rewrites the following instructions:

- The `mov`, `push`, and `pop` instructions that manipulate segment registers (`%cs`, `%ds`, `%es`, `%fs`, `%gs`, and `%ss`).
- The `call`, `jmp`, and `ret` instructions that cross a segment boundary.
- The `iret` instruction.
- The instructions that manipulate special registers, such as the task register.
- The instructions that read and write the flag register.

3.1.2 Rewriting with subroutine calls

In the insertion method, the partial emulator has to issue four system calls for each execution of a privileged or non-privileged instruction to be detected, as described in Section 3.1.1. We eliminate this overhead by rewriting the privileged and non-privileged instructions with subroutine calls that emulate these instructions. An example of rewriting follows:

```
Before:
        mov     %eax, %cr3
After:
        call    mov_eax_cr3
```

The subroutine `mov_eax_cr3` performs emulation of the instruction. We also perform inlining for simple instructions. Our newer partial emulator consists of 800 lines of C code in the different address space, and 250 lines of C code and 300 lines of assembly language code in the same address space as a user-level OS.

3.2 Redirection of system calls and page faults

When a user process on a user-level OS issues a system call or causes a page fault, the host OS should not interpret the event by itself. Instead, the host OS should notify the user-level OS of the event (a system call or page fault). If the host OS provides a redirection mechanism, this is an ideal mechanism for a user-level OS (Figure 3 (a)).

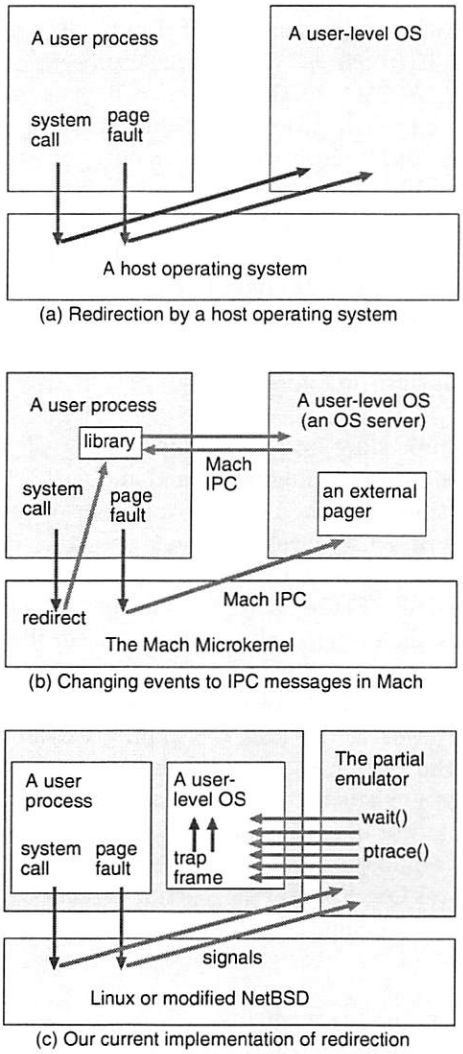

Figure 3: Redirection of system calls and page faults.

The Mach microkernel [GDFR90] includes a redirection mechanism of system calls for executing Unix binaries. When a user process (task) of Unix issues a system call, the Mach microkernel sends back a message to the same task (Figure 3 (b)). The user task includes the support module that receives the message from the microkernel, and performs an RPC to the remote Unix server. The Mach microkernel has a similar mechanism for handing page faults. This mechanism is called an external pager [GDFR90].

To implement the system call redirection, we use

the process trace facility of Linux at first. In Linux, if a parent (or tracing) process specifies `PTRACE_SYSCALL` to the system call `ptrace()`, the child (or traced) process continues execution as for `PTRACE_CONT`[1], but it will stop on entry or exit of the system call.

When the child process is stopped, the parent process can examine and modify the CPU registers and arguments or results of the system call. Solaris and other Unix System V also have a similar facility through the /proc filesystem.

The procedure for redirecting system calls is similar to that for handing privileged and non-privileged instructions described in Section 3.1.1. When a user process of a user-level OS issues a system call, the process of the host OS is stopped on entry of the system call (Figure 3 (c)). The partial emulator changes the system call number with an illegal one and continues the execution. Next, the host OS tries to execute the body of the system call in a regular way. However, the host OS cannot execute it because the system call number is wrong. Therefore, the host OS sets an error value and notifies the partial emulator of exiting of the system call. Next, the partial emulator prepares an interrupt frame on the user-level OS. Finally, the partial emulator changes the program counter and switches to the user-level OS.

Page faults (`SIGSEGV`) are handled in a similar way as system calls. A difference is that the partial emulator has to send a signal to get values of some control registers when the host OS is Linux. In the partial emulator described in Section 3.1.2, the signals (`SIGSEGV`) are handled by the partial emulator code in the user-level OS. Therefore, we can reduce the overhead of context switches between the user-level OS and the partial emulator.

3.3 Emulation of Memory Management Unit

In IA-32, operations of MMU (Memory Management Unit) are performed thorough the register cr3 (Control Register 3). IA-32 uses two-level page tables. The register cr3 holds the physical address of the top level page table called *Page Directory* [Int97].

To emulate MMU and build address spaces for the user-level OS and its user processes, we use the system calls `mmap()` and `munmap()`. When the content of the register cr3 is changed, the partial emulator first compares each entry of the new page table with that of the previous page table. If a new page table entry no longer has a page, the partial emulator unmaps the corresponding page with the system call `munmap()`. If a new page table entry has a new page, the partial emulator maps the page with the system call `mmap()`. Otherwise, the partial emulator does nothing.

[1] `PT_CONTINUE` in BSD.

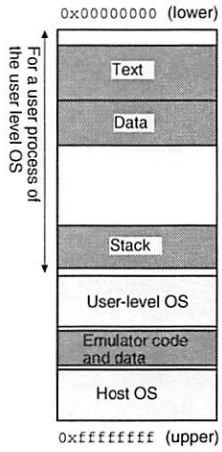

Figure 4: The address space of the user-level OS.

To emulate MMU with the system calls `mmap()` and `munmap()`, we have to solve the following problem in the insertion method (Section 3.1.1). These system calls should be invoked by the process of the user-level OS. These system calls manipulate the address space of the issuing process itself, and the parent or tracing process cannot manipulate the child or traced process.

To solve this problem, we embedded a support module in the address space of the user-level OS. Figure 4 shows the address space of the user-level OS. The lower regions of the address space are for user processes of the user-level OS (Text, Data, and Stack). The upper end is used by the host OS, and cannot be used by the user-level OS. Below the host OS region, there is a region for the user-level OS. We allocate a space for the `mmap/munnap` module between the host OS region and the user-level OS region.

When the partial emulator detects the manipulation of MMU (setting a value to the register cr3), the partial emulator compares the old and the new

page table. Based on the differences between the old and the new page table, the partial emulator makes an issuing plan of the system calls `mmap()` and `munmap()`, and stores the plan to the region of the emulator code and data in the address space of the user-level OS (Figure 4). After that, the partial emulator changes the program counter of the child process (the process executing the user-level OS) to the code of the partial emulator in the address space of user-level OS, and switches to the user-level OS. In the user-level OS, the emulator code issues the system calls `mmap()` and `munmap()` according to the plan. The partial emulator (the parent process) does not intercept these system calls and allows passing them to the host OS. Finally, the code executes a special instruction (`int $3`) to switch to the partial emulator. The partial emulator changes the program counter to the next instruction of the MMU operation, and continues execution.

We have described the procedure for the insertion method (Section 3.1.1). In the rewriting method (Section 3.1.2), MMU emulation is performed by the subroutines in the same address space as the user-level OS. The partial emulator in the separated address space does nothing about MMU emulation.

In both procedures, comparing the old and the new page table and issuing the system calls `mmap()` and `munmap()` are heavy tasks, and they are big sources of performance degradation. We will show the experimental results about MMU emulation in Section 6.

3.4 A console, a timer and a disk

The current partial emulator provides minimum peripheral devices: the keyboard for input, the video RAM for console output, and the timer for periodic interrupt.

For persistent storage, we have developed a small device driver that is derived from the memory disk driver of NetBSD. The memory disk of NetBSD is a block device, and it does not use interrupt for I/O. Instead, the memory disk reads and writes a memory reason in the kernel. We have changed these operations with the partial emulator calls (Section 3.1.1) or the system calls for the host OS (Section 3.1.2).

4 Modifying NetBSD for hosting the partial emulator

At first, we settled our goal to implement our partial emulator to run NetBSD on Linux. We chose NetBSD because it supports many hardware architectures, and each architecture-specific part seems small. We chose Linux because Linux already had User Mode Linux, so it was obvious that Linux has enough facilities for running a user-level OS. However, porting the i386-dependent part of NetBSD to the Linux architecture was not easy for us. Therefore, we have developed the techniques with partial emulation and rewriting of machine instructions.

After we had succeeded in running NetBSD on Linux, we started running NetBSD on NetBSD. We found that the unmodified NetBSD does not provide enough facilities to implement our partial emulator. The *ktrace* facility of NetBSD can be used to record events of entering and exiting of system calls. However, the ktrace facility does not allow stopping traced processes and changing registers and memory.

To run NetBSD on NetBSD, we decided to add a new facility to the host NetBSD kernel. The basic task is to introduce the `PTRACE_SYSCALL` facility of Linux to NetBSD. In the following subsections, we will show our modifications to NetBSD 1.6.1 in detail.

4.1 Modifications of the core part of NetBSD

We have modified the following three files in the core part of NetBSD (the architecture-independent part of NetBSD):

- `sys/proc.h`
- `sys/ptrace.h`
- `kern/sys_process.c`

We have added the following flag to the `struct proc` in `proc.h`:

```
#define P_SYSTRACED 0x800000
   /* System call is traced. */
```

We have added the following request for the system call ptrace() in the file ptrace.h:

```
#define PT_SYSCALL 12 /* Continue and stop
  at the next (return from) syscall. */
```

The flag and the request are used by the function sys_ptrace() of sys_process.c in the core part and the function syscall_plain() of syscall.c in the i386 architecture-specific part.

To the function sys_ptrace() in sys_process.c, we have added some lines for the request PT_SYSCALL next to the request PT_CONTINUE. The essential difference between PT_SYSCALL and PT_CONTINUE follows:

```
if (SCARG(uap, req) == PT_SYSCALL) {
  SET(t->p_flag, P_SYSTRACED);
} else {
  CLR(t->p_flag, P_SYSTRACED);
}
```

If the request for the system call ptrace() is PT_SYSCALL, we set the flag P_SYSTRACED to the traced process. Otherwise, we clear the flag.

In addition to the above modification, we have defined a new function as follows:

```
process_systrace(p)
        struct proc *p;
```

This function is called from the entry point and the exit point of the system call when the flag P_SYSTRACED is set. This function stops the current process (the traced process) as if it would receive the signal SIGTRAP. If the tracing process is sleeping typically by issuing the system call wait(), the current process wakes up the tracing process.

4.2 Modifications of the i386-dependent part of NetBSD

We have modified the following three files in the i386-specific part of NetBSD:

- arch/i386/i386/syscall.c

```
void syscall_plain(frame)
  struct trapframe frame;
{
  ...
  p = curproc;
  if (ISSET(p->p_flag, P_SYSTRACED)) {
    CLR(p->p_flag, P_SYSTRACED);
    process_systrace(p);
  }
  code = frame.tf_eax;
  callp = p->p_emul->e_sysent;
  ...
  code &= (SYS_NSYSENT - 1);
  callp += code;
  ...
  error = (*callp->sy_call)(p, args, rval);
  ...
  if (ISSET(p->p_flag, P_SYSTRACED)) {
    CLR(p->p_flag, P_SYSTRACED);
    process_systrace(p);
  }
  userret(p);
}
```

Figure 5: Checking of the P_SYSTRACED flag on entry and exit of the system call.

- arch/i386/i386/process_machdep.c
- arch/i386/include/reg.h

We have inserted the invocation of the function process_systrace() to the function syscall_plain() in the file syscall.c, as shown in Figure 5. This function fetches the system call number in the register eax and calls the body of the system call by consulting the jump table in the p->p_emul->e_systent. Before getting the system call number from the register eax, we check if the flag P_SYSTRACED is set. If it is set, we call the function process_systrace() which is described in Section 4.1. The same function is also called at the end of the function before returning to the user mode.

Moreover, we have changed the files process_machdep.c and reg.h, and extended struct reg for the PT_GETREGS request of the system call ptrace(). In addition to regular registers for debugging, we need other values in control registers and the trap frame. For example, the register cr2 in PCB (Process Control Block) and the trap number are needed by the partial emulator. In Linux, we used a signal facility to get these values. If we set the program counter to an illegal address, the process receives a signal. At this time, the values that are needed by the partial emulator are pushed on the stack. In NetBSD, we did not use a signal facility. Instead, we extended struct reg in the request PT_GETREGS of the system call

`ptrace()`.

5 Modifications to NetBSD and FreeBSD for running as user-level operating systems

In the implementation of a user-level OS, the final goal is to generate the user-level OS from the corresponding native OS for the bare hardware automatically. However, we had to slightly modify the native NetBSD and FreeBSD. In this section, we show the modifications to NetBSD and FreeBSD for running them as user-level OSes.

5.1 Modifications to NetBSD for running as a user process

We ran NetBSD 1.5.2 as a user process by changing 6 constants to adjust the address space and removing device drivers from the configuration file. The base address of NetBSD is changed from `0xc0000000` to `0xa000000` because the memory region after `0xc0000000` is occupied by the host operating system kernel.

Note that this modification is achieved without detailed knowledge about the NetBSD kernel. This is the significant difference from conventional user-level OSes, such as User Mode Linux. User Mode Linux requires adding a new architecture called *um*. The code size under the `um` directory is 33,000 lines, and this is comparable with the code size of the native i386 architecture (44,000 lines). This porting may cause a maintenance problem. When the native i386 architecture gets a new facility, the *um* architecture has to catch up the facility manually. In contrast, the core of our user-level NetBSD is automatically generated from the native i386 NetBSD. Therefore, we can follow the evolution of native i386 NetBSD more easily.

5.2 Modifications to FreeBSD for running as a user process

We have also executed the FreeBSD 4.7 kernel as a user process. In addition to address constants, we have changed the places that call BIOS. We have simply commented out the places and replaced with the code that returns parameters, such as the size of memory and the type of CPU. Since we did not have BIOS code, changing the FreeBSD kernel was much easier than implementing BIOS. Furthermore, changing the kernel reduces the effort to implement the partial emulator. Calling BIOS requires emulation of *the virtual 8086 mode* of Pentium. Our partial emulator does not have that facility. If we have BIOS code and a more powerful emulator, we do not have to modify these places.

6 Performance

We made experiments to measure the performance of our user-level OS (NetBSD 1.5.2). In this section, we show the results of microbenchmarks and an application benchmark.

All experiments were performed on a PC with a Pentium III 1GHz and 512M bytes of main memory. The host operating system for our partial emulator is Debian 3.0 with the Linux kernel 2.4.20 [2].

6.1 Microbenchmarks

As microbenchmarks, we use the following user programs:

`loop`: This program increments a variable in a loop. This program does not issue any system call during the experiments although the execution is interrupted by the timer.

`getpid`: This program issues the system call `getpid()`, repeatedly.

`pipesw`: This program creates two processes which are connected with two pipes. Each process writes and reads a byte for each step in a loop, so two context switches and four system calls are performed in a step.

`fork`: This program creates and terminates processes repeatedly. The parent process issues the system calls `fork()` and `wait()`, and the child processes issue the system call `exit()`.

[2] Currently, NetBSD on NetBSD is not stable enough to measure performance.

Table 1: The execution times of microbenchmark programs.

OS/Environment	program loop (n sec)	getpid (u sec)	pipesw (u sec)	fork (m sec)
NetBSD/PE-Insert/Linux	2.04	136	2880	89.9
NetBSD/PE-Rewrite/Linux	2.00	23.0	1030	19.0
NetBSD/Physical	1.99	0.360	19.8	0.380
NetBSD/Bochs/NetBSD	288	68.0	1600	34.9
NetBSD/VMware/Linux	2.01	3.53	83.7	2.550
Linux/Physical	1.99	0.299	5.53	0.114
User Mode Linux/Linux	1.99	44.1	665	31.7
Linux/Plex86/Linux	1.99	20.0	346	1.76

In these experiments, we measured the peak performance. In each execution, the task is repeated from 100 to 10,000,000 times. The number of trials is determined to be high enough to reach several tents of milliseconds to several seconds. The execution times were obtained using the system call `gettimeofday()` for the host OS, and divided by the number of iterations.

The result is shown in Table 1. In Table 1, "PE-" sands for our partial emulator. "Insert" means the insertion method (Section 3.1.1), and "Rewrite" means the rewriting method (Section 3.1.1), respectively. For reference, we include the results of the following operating systems and environments:

- NetBSD 1.6.1 on the physical PC
- NetBSD 1.5.2 on Bochs 2.02 on NetBSD 1.6.1
- NetBSD 1.5.2 on VMware 4.0 on Linux 2.4.20
- User Mode Linux 2.4.20-uml-6 on Linux 2.4.20
- Linux 2.4.20 on the physical PC
- Linux on Plex86 2003-02-16 on Linux 2.4.20 with NFS [3]

For the program `loop`, both of our partial emulators ("PE-Insert" and "PE-Rewrite") produced almost same performance as the physical PC because the measured part of the benchmark program is executed by the real CPU directly. By the same reason, our user-level NetBSD is faster than NetBSD on Bochs by a factor of 100.

[3]The current Plex86 uses a small RAM disk as a root device. We mounted the host file system with NFS and ran the benchmarks after changing the root directory to the host root.

In the cases of `getpid`, `pipesw` and `fork`, our partial emulator is slower than the physical machine by a factor of 100 and VMware by a factor of 10. This slowdown is cased by overheads of the system call redirection (Section 3.2) and the MMU emulation (Section 3.3). We got performance improvement by a factor of 2.8 to 5.9 between "PE-Insert" and "PE-Rewrite".

We cannot simply compare our partial emulator with User Mode Linux and Plex86 because the user-level OSes are different. As shown in Table 1, NetBSD/Physical is slower than Linux/Physical in those microbenchmarks. If we ignore the difference of user-level OSes, NetBSD on our partial emulator of "PE-Rewrite" is faster than User Mode Linux in the cases of `getpid` and `fork`. Our partial emulators are slower than Plex86 because Plex86 uses an efficient hardware mechanism called PVI, as described in Section 2.

6.2 An application benchmark

We ran the make command for compiling the GNU patch command (Version 2.5.4), and measured the execution times. The source code of the patch command consists of 15 C files and 17 headers. Total length of those files is 9200 lines or 244 k bytes [4]. The result is shown in Table 2.

NetBSDs on our partial emulators were faster than NetBSD on Bochs by a factor of 10. However, they were slower than NetBSD on the physical PC

[4]Although the source files are same on NetBSD and Linux, the header files in `/usr/include are` different. In this compilation, total 460 header files (2 M bytes) are included in NetBSD while 770 header files (6 M bytes) are included in Linux. Therefore, the execution time on NetBSD/Physical is shorter than that on Linux/Physical.

and VMware by a factor of 15 and 4, respectively. The ratios are smaller than the results of the microbenchmarks `getpid`, `pipesw`, and `fork` because this application benchmark includes a CPU workload. Our user-level NetBSD is slower than User Mode Linux and Linux/Plex86 because of the MMU emulation overhead.

7 Future directions

Our projects began on June 2002, and we have many tasks to be accomplished. Those tasks are classified into two categories:

- Adding new functions.
- Improving performance.

For each of functionality and performance, we have to choose or combine the following strategies:

- To modify the partial emulator.
- To modify host OSes.
- To modify user-level OSes.

The first strategy is best because it is independent of host and user-level OSes.

The first priority task on functionality is to add a networking facility. We are implementing a pseudo serial device for communicating between a user-level OS and a host OS. This serial device can be used for passing PPP packets. We also have a plan to implement an Ethernet-like device.

Table 2: The execution times of compilation in seconds.

OS/Environment	make (sec)
NetBSD/PE-Insert/Linux	52.5
NetBSD/PE-Rewrite/Linux	13.7
NetBSD/Physical	3.6
NetBSD/Bochs/Linux	550
NetBSD/VMware/Linux	3.9
Linux/Physical	4.1
User Mode Linux/Linux	9.5
Linux/Plex86/Linux	13.0

We are also interested in a function to access host file systems, as similar to *the host file system* of User Mode Linux. A straightforward implementation method is to insert a module to the VFS layer while we have to implement the module for each host OS. By using the networking facility, we can access host file systems through the NFS protocol and the SMB protocol.

As shown in Section 6, the main sources of overhead are the MMU emulation and the system call/page fault redirection.

To enhance the MMU emulation, we can cache address spaces as user processes of the host OS. As similar to the hardware context table of SPARC [SPA92], we can preserve and reuse user processes as page tables. In other words, the partial emulator forks when the MMU register of the page table gets a new value. If we cache page tables, we have to discard unused page tables or user processes of the host OS. If some LRU algorithm works well, we do not have to modify the user-level OS. Otherwise, we should modify the user-level OS to invalidate the cache on termination of its user processes.

We are also studying to introduce a new kernel level abstraction called *a virtual page table*. With this facility, a user-level OS can manipulate its page tables by issuing a new system call for the host OS. Unlike regular system calls, this system call for virtual page tables should depend on underlying hardware because we can preserve the structure and semantics of the base native OS. If we use a different facility, such as the external pager of the Mach microkernel, we have to change the base native OS more.

In the current implementation, the partial emulator does not handle the segment facility of IA-32 completely because most operating systems including NetBSD do not make use of the segment facility. The partial emulator interprets only address translation and write protection in the two-level page table. The partial emulator does not interpret other bits, such as Accessed and User/Supervisor for performance. Therefore, a user-level OS cannot perform page replacement efficiently. Moreover, the partial emulator does not change the memory protection on switching from the kernel mode to the user mode for performance reason. In IA-32, most operating systems including NetBSD do not change the MMU setting when the context is transferred from the user mode to the kernel mode or vice versa. Therefore,

user processes can access the kernel memory. We would like to add a protection facility of the kernel memory after implementing the caching facility.

8 Conclusion

In this paper, we have proposed an implementation method of user-level operating systems based on partial emulation of PC hardware and rewriting of machine instructions at compile time. Unlike conventional methods, user-level operating systems are generated from the native operating systems. Therefore, no detailed knowledge about the native operating systems is needed to implement the user-level operating system. Based on the proposed method, we have executed NetBSD and FreeBSD kernels as user processes on Linux and NetBSD with small changes from the corresponding native systems.

The partial emulator on Linux can be used for running NetBSD and FreeBSD applications on Linux. We have nested operating system environments for NetBSD, so we can use several versions of NetBSD co-exist on the same platform.

We hope that our partial emulator will be one of the most popular tools for nested BSD operating systems. For developers, nested operating systems will be an essential facility to experiment with new kernels or new release while keeping the base environment safely. We are working on adding a networking facility to our partial emulator. With the networking facility, we can execute Internet servers on user-level operating systems, and we can couple user-level operating systems with the host operating system more tightly.

References

[AAS94] P. Ashton, D. Ayers, and P. Smith. SunOS Minix: a tool for use in operating system laboratories. *Australian Computer Science Communications*, 16(1):259–269, 1994.

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *the 4th Annual Linux Showcase & Conference*, 2000.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation*, pages 137–151, 1996.

[GDFR90] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[HH79] E. C. Hendricks and T. C. Hartmann. Evolution of a virtual machine subsystem. *IBM System Journal*, 18(1):111–142, 1979.

[HMM03] Brian N. Handy, Rich Murphey, and Jim Mock. *FreeBSD Handbook, Linux Binary Compatibility*, 2003.

[Int97] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 1997.

[Law03] Kevin P. Lawton. *The Plex86 x86 Virtual Machine Project*, 2003. `http://plex86.sourceforge.net/`.

[LDG+03] Kevin Lawton, Bryce Denney, N. David Guarneri, Volker Ruppert, Christophe Bothamy, and Michael Calabrese. *Bochs x86 PC emulator Users Manual*, 2003. `http://bochs.sourceforge.net/`.

[LF03] Federico Lupi and The NetBSD Foundation. *The NetBSD Operating System, A Guide, Chapter 14. Linux emulation*, 2003.

[LW73] Hugh C. Lauer and David Wyeth. A recursive virtual machine architecture. In *Proceedings of the ACM SIGOPS/SIGARCH workshop on virtual computer systems*, pages 113–116, 1973.

[Pap00] A Connectix White Paper. *The Technology of Virtual PC*, 2000.

[SPA92] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, 1992.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, 2001.

[Tad92] Yoshikatsu Tada. A virtual operating system VXinu — its implementation and problems. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J75-D-1(1):10–18, 1992.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, 1992.

A Digital Preservation Network Appliance Based on OpenBSD

David S. H. Rosenthal
Stanford University Libraries
Stanford, CA 94305
http://www.lockss.org

Abstract

The LOCKSS program has developed and deployed in a world-wide test a system for preserving access to academic journals published on the Web. The fundamental problem for any digital preservation system is that it must be affordable for the long term. To reduce the cost of ownership, the LOCKSS system uses generic PC hardware, open source software, and peer-to-peer technology. It is packaged as a "network appliance", a single-function box that can be connected to the Internet, configured and left alone to do its job with minimal monitoring or administration. The first version of this system was based on a Linux boot floppy. After three years of testing it was replaced by a second version, based on OpenBSD and booting from CD-ROM.

We focus in this paper on the design, implementation and deployment of a network appliance based on an open source operating system. We provide an overview of the LOCKSS application and describe the experience of deploying and supporting its first version. We list the requirements we took from this to drive the design of the second version, describe how we satisfied them in the OpenBSD environment, and report on the initial deployment of this second version of the appliance.

1 Introduction

The LOCKSS[1] (Lots Of Copies Keep Stuff Safe) program has developed and deployed test versions of a system for preserving access to academic journals published on the Web. The fundamental problem for any digital preservation system is that it must be affordable for the long term. To reduce the cost of ownership, the LOCKSS system uses generic PC hardware, open source software, and peer-to-peer technology. It is packaged as a "network appliance", a single-function box that can be connected to the Internet, configured and left alone to do its job with minimal monitoring or administration. The appliance has to operate, exposed to the Internet, in environments lacking skilled system administrators, without imposing large administrative costs to install, maintain or upgrade it.

The first version was based on a boot-floppy distribution of Linux. After three years of testing at over 50 libraries world-wide, this appliance level of the system was replaced by a second version, based on a modified version of the OpenBSD install CD-ROM. It was deployed to the test systems around the world in January 2003. The application levels of the system have also been redesigned and reimplemented from scratch; deployment of this new implementation started in May 2003.

We focus in this paper on the design, implementation and deployment of a network appliance based on an open source operating system. The goals and overall architecture of the LOCKSS system [18], the redesign of the protocol by which the peers communicate [12] and its economic underpinnings [19] are covered elsewhere.

We provide an overview of the application the network appliance is designed to support. We describe the experience of deploying and supporting its first version. We list the requirements we took from this as a basis for the design of the second version, describe how we satisfied them in the OpenBSD environment, and report on the initial deployment of this second version of the appliance.

[1] LOCKSS is a trademark of Stanford University.

2 The LOCKSS Program

Scientific communication has transitioned to the Web. In particular, much peer-reviewed science now appears only in e-journal form [10]. Academic journals are funded by university and other librarians paying institutional subscription rates. These librarians consider it part of their job to preserve access to the record of science for future generations. The transition to the Web has meant a transition from a purchase model, in which librarians buy and own a copy of the journal, to a rental model, in which librarians rent access to the publisher's copy. Year-by-year rental provides no guarantee of future access, and librarians fear the worst. Publishers are motivated to allay these fears and persuade libraries to switch to electronic-only subscriptions, which save the publishers money.

The LOCKSS program is implementing the purchase model for the Web, providing tools librarians can use to take custody of, and preserve access to, web-published journals. The tools allow libraries to run persistent web caches that:

- *collect* material by crawling the e-journal Web sites,
- *distribute* material by acting like a proxy cache to make it seem to a library's readers that web pages are still available at their original URL, even if they are no longer available there from the original publisher [21].
- *preserve* material by cooperating with other library's caches in a peer-to-peer network to detect and repair damage.

The LOCKSS cache is implemented as a single daemon process written in Java. It includes a specialized web crawler, the Jetty [6] web server to provide both a Web proxy and an administration user interface, and the peer communication protocol.

We believe that the major threats to digital preservation are economic rather than technological; library budgets, especially for preservation, are never adequate [4]. If LOCKSS is to succeed in preserving access for future generations, libraries must be able to afford the system in the long term, through the inevitable ups and downs of their budgets. The goal of the LOCKSS program is to reduce the economic risks by spreading the total cost of preservation across many independent budgets, minimizing the impact on each individual budget, and lowering the economic barrier to entry.

Cost reduction has, therefore, been a major focus of the program from its inception. Both the LOCKSS daemon and its operating system platform are free and open source, and use generic PC hardware to reduce hard costs as much as possible.

Soft costs, especially support and system administration, can easily dominate the hard costs. Maintaining availability of Internet services over the long term is very difficult. Even at sites with expensive, skilled professional system adminstrators between 20-50% of outages are caused by operator error [15].

Although it would be possible to include the LOCKSS daemon among a variety of services on a large server, the very long time horizons and slow operation of the system are unlike the other services that it might be running.

Machine boundaries can provide effective fault isolation. This encourages our belief that the overall reliability and cost of the system can be improved by packaging the daemon, its system infrastructure, and the hardware it needs into a network appliance. By reducing the economic barrier to entry as far as possible we hope to broaden the base of libraries engaged in digital preservation activities. On the other hand, we expect many larger libraries will run the LOCKSS daemon without its appliance platform.

Our network appliance design is intended to reduce both the cost and the risk of system administration by identifying the expensive and risky operations involved, and either eliminating or automating them. Our top candidates are installation, upgrade, and recovery from compromise. We have succeeded in almost completely eliminating system administrator involvement in all three. We believe our ideas and experience in this area could be helpful to others.

3 Lessons

The first version of the LOCKSS appliance [18] was based on the Linux Router Project (LRP) platform [16], a boot floppy containing a minimal but functional Linux system in a RAMdisk. For our

application, it was capable of downloading and installing into a temporary file system the LOCKSS daemon and the software on which it depended, such as the Java Virtual Machine (JVM), that would not fit on the floppy,

To begin running the appliance, the host institution downloaded and ran a Windows program that formatted, wrote and checked a generic version of the floppy. When a generic PC was booted from this floppy, it asked the operator for the necessary configuration information then personalized the floppy, partitioned the disk and created the necessary file systems on it.

Booting the personalized floppy ran the normal LRP boot sequence then downloaded the necessary LOCKSS software, installed it in a temporary file system, and finally invoked the JVM to run the daemon.

The system evolved over about 3 years of testing to run at over 50 libraries worldwide and was generally sucessful in requiring neither great skill nor much attention from the host institution. This taught us many valuable lessons. The most important of these was that running the system exclusively from write-locked media (and from software verified against hashes on write-locked media) greatly simplifies two critical tasks: installing and configuring a new system, and recovering from a compromise.

In unskilled hands the process of installing and configuring a Unix system to be adequately secure is a daunting and error-prone task. Running an almost completely pre-configured system from write-locked media obviates almost all the effort and risk.

Restoring a compromised system is a daunting and error-prone task even in skilled hands. A simple reboot is all that is needed to restore the LOCKSS appliance to a known state. If the compromise damaged or even completely destroyed the stored content, the normal peer-to-peer communication will compare the damaged content with the content at other peers and repair it from them or the publisher.

It may take many weeks to completely recover a destroyed peer. However, once the damage has been detected any potentially damaged and not yet repaired content actually requested by readers can be obtained from other peers. Old academic papers are infrequently accessed [3], so the load of these requests is not significant.

Other important lessons we learned included:

- The floppy disks, used only during boot, were remarkably reliable. We had very few media problems.
- Operators did not always pay attention to instructions to write-lock their floppy disks.
- Squeezing the software we needed into the limited confines of even a 1.68MB-formatted floppy, and working with non-standard floppy formats, was too time-consuming and error-prone for our small team.
- In the interests of security, MD5 hashes of the downloaded platform software were stored on the floppy. It was thus necessary to persuade each test site to create a new boot floppy before it could run a newly-released version of the platform software, for example to patch a vulnerability. This had its bright side, in that we were sure their configuration was consistent, but it was an operations nightmare and very time-consuming.
- Partly because of this inefficient upgrade path, the small number of security vulnerabilities discovered in our environment during the test consumed a totally disproportionate amount of the team's efforts. This was compounded by the way they tended to occur during holidays.
- A second reason for the difficulty in responding to security vulnerabilities was the need to rebuild the distribution after patching it. LRP's build process was never very robust and our modifications were so extensive that building it became too fragile a process to manage under the time pressure of a security incident.
- LOCKSS caches need static, globally routable IP addresses and communicate via UDP. They can be run behind firewalls or network address translation (NAT) boxes but doing so involves negotiation with the host institution's network administrators. If they are run outside the firewall, a different negotiation with the network administrators may be required. Some sites insist on security audits before installation. The LOCKSS appliance needs a simple, easy-to-explain, and convincing security story.

4 Requirements

From this experience we developed requirements for the second version of the system:

- Use only generic PC hardware.
- Install the operating system, application and all other software afresh in a newly-created evanescent filesystem on every boot.
- Even if the system is compromised, there must be no place a Trojan horse could be hidden. All file systems that persist across reboots must be mounted with *noexec*, *nosuid* and *nodev* options.
- All software, including the daemon, must either run directly from read-only media or from packages whose signatures are verified by software running directly from read-only media before being installed into evanescent file systems.
- The public keys used to verify the signatures, and thus permitted to sign software, must be stored on write-locked media.
- Each package should carry multiple signatures. It must be impossible for revocation of a single key to cause the system to fail.
- The set of keys trusted to sign packages must be under the control of the appliance operator.
- Media containing system configuration data or keys must be write-locked while the network is up.
- Non-writability of media must be tested, not assumed.
- No process sending or receiving network data may run as root.
- The system must walk the user through the configuration process and test that the supplied values work before accepting them.
- Upgrading the system should require only a reboot.
- The response to a newly-discovered vulnerability should be simple and rapid, both to remove the vulnerability and repair any damage to a compromised appliance's state.
- The system must be easy for a small team to support. Our criteria for this were:
  - Use a major OS distribution with security as its primary focus; don't use a minor distribution with a limited base of support such as LRP.
  - Don't use a floppy-based distribution such as PicoBSD [5]; squeezing GnuPG and other things we need onto a floppy is not feasible, and the industry is phasing out floppy disks.
  - Don't change the OS build-from-source process.
  - Maintain a minimal footprint in the OS source.
  - Build the entire environment from scratch automatically every night.

5 Trust Model

We assume that libraries are capable of maintaining the physical security of their LOCKSS appliances. This leads us to trust to some extent the content of write-locked media in the machine's drives; an attacker capable of replacing the CD from which the machine boots can obviously damage the machine's content.

We limit this trust to the boot image and a few other files on the CD that wouldn't fit into it (Section 6). All other software must carry a valid signature from at least one of a set of keys which are trusted by the system's administrator and not known to be revoked. The trusted software consists of the kernel and the utilities needed to perform the key revocation check, signature validation and software installation.

This technique has similarities to the Trusted Computing Platform Architecture (TCPA) [2]. Our goals, however are quite different. TCPA is intended to enable programs which do not trust the administrator of the system on which they are running to verify that the system's integrity is attested to by keys that the program does trust. LOCKSS peers do not trust each other and do not run third-party software; the integrity of the software on a LOCKSS cache is of purely local interest. Thus our, much weaker, goal is to assure the administrator of the

system that, at least immediately after a reboot, it is running only software whose integrity is attested to by keys that the adminstrator trusts.

Lacking the hardware and BIOS support of TCPA, we cannot fully achieve this goal. A remote root compromise could in some circumstances allow the attacker to modify the system's BIOS and thereby disable the signature verification process. This in turn might allow spurious software to persist across reboots. To minimize this and other risks of a remote root compromise, we run with OpenBSD's *kern.securelevel* variable set to 2, the most restrictive possible, and we remove the debugger from our kernel.

6 Background

These requirements led us to a design based on adapting the OpenBSD install CD. The design of this bootable CD is a series of layers, which we describe from the outside in:

- Booting a PC from a CD requires that the image of a 2.88MB "boot floppy" be present on the CD.
- In the case of OpenBSD, this "boot floppy" contains an FFS file system and the *biosboot* program, which the PC's BIOS locates and starts.
- The file system contains a compressed kernel image, which *biosboot* loads into memory and executes.
- Part of the kernel's data space is a RAMdisk image containing an FFS file system 1.7MB big, which is mounted as the root file system.
- This file system contains skeletal system directories such as *etc* and */dev*. It also contains a single *crunched* binary which implements all the commands needed for the installation.

The kernel starts up in the normal way, but it runs a special */etc/rc* script that walks the user through the installation process. The shell and all the other commands needed to run this script are in the crunched binary.

The crunched binary is constructed from the regular source tree by a pair of tools called *crunchgen* and *crunchide*. Crunchgen works from a list of commands by locating the Makefile for each command in the source tree, and writing a new Makefile, a set of stub programs and a top-level *main()* that calls the stub chosen by *argv[0]*. The new Makefile builds the appropriate set of object files from the command sources, then links each command and its stub into an intermediate file that is processed by crunchide to hide all its global symbols except those of the stub. Finally, the intermediate files are linked with the generated *main()* into the crunched binary.

7 Design

Our approach was to replace the */etc/rc* script run by the minimal "boot floppy" environment on the install CD with a slightly augmented regular system's */etc/rc* which:

- Establishes swap space.
- Confirms that the configuration floppy is not writable.
- Creates an evanescent file system in the swap space; redirecting the system directories into it via symlinks.
- Validates the signatures on the necessary software packages.
- Installs the packages into the evanescent file system (via the symlinks).
- Installs the network configuration information from the floppy.
- Runs only the essential services:
  - The SSH daemon [22] for remote administration, with privilege separation [17]
  - The LOCKSS daemon inside the JVM run as an unprivileged user.
  - The appliance does not run sendmail. Outbound mail is sent by ssmtp, a minimal SMTP client. Inbound mail is not accepted.
- Installs a *crontab* entry that implements the automatic package update mechanism.

The package update mechanism at intervals chooses at random from a list of download servers. As an unprivileged user it checks that server for new packages and signatures and downloads any found. It caches them on the hard disk. They will be used at the next reboot if their signatures are valid. Although a compromise could allow these cached packages to be modified, doing so would invalidate their signatures. An invalid signature causes the boot process to ignore that version of the package and revert to an earlier version, probably the one on the CD.

8 Build Process

The implementation resides entirely in directory hierarchies below */usr/src/distrib/i386/lockss/*. except for:

- The LOCKSS kernel config file.
- Three directories in */usr/src/distrib/special/*:
  - A skeleton implementation of *host*, to avoid the full glory of the BIND implementation.
  - A skeleton implementation of *sudo*, capable only of being used by root to give up privileges. This allows us to avoid root processes accessing the network during the boot sequence.
  - An altered Makefile for *init*, needed to get it to go multi-user by default.
- An additional entry in */usr/src/distrib/i386/Makefile* that builds the LOCKSS CD.

Our goal of maintaining a small footprint in the distribution source has been met.

Our build script starts by checking the patch branch out from OpenBSD's AnonCVS [7] service into a temporary build tree, updating our copy of the ports tree, then checking the implementation of the LOCKSS appliance platform out from our CVS server and copying it into place in the build tree.

The build process then patches the kernel source to install one additional driver, for *swdt*, a kernel-based software watchdog that reboots the system if a user-level daemon fails to reset the watchdog at least every 30 seconds. It can sometimes recover a system that has locked up.

The build process is essentially the OpenBSD install CD build process. It results in a bootable CD very similar to OpenBSD's, but containing in addition to the basic OpenBSD packages a set of packages from the ports tree, including the JVM, the Red Hat emulator needed to run it, ssmtp and some libraries they depend on. It also contains a set of LOCKSS packages containing the daemon and its environment. Installed on the CD itself is GnuPG [9], and the shared libraries it needs to run.

Distributing new CD images, having administrators burn them and use them to reboot their systems is too time-consuming and expensive to fix security vulnerabilities or for routine upgrades to the application. The *crontab* entry described above can download packages and cache them on the disk for use at the next reboot. To avoid mistakes, there is no separate build process for packages distributed in this way. An upgrade package is built by building an entire CD image, extracting the package from it and signing it. The package and its signature are placed on the download servers, and mail is sent to the sites asking them to:

- log in as root
- execute a script that checks for new packages
- reboot

9 Implementation

The implementation consists almost entirely of three shell scripts, residing in the RAMdisk image in the CD's "boot floppy", which are executed during the system boot process. The work they do means that our appliance takes a rather long time to boot, but in the field of digital preservation speed is not a requirement (see 10). The scripts are:

- */etc/rc.0.lockss*, which is executed very early in */etc/rc*, before swap is enabled;
- */etc/rc.1.lockss*, which is executed later in */etc/rc*, once the file systems are mounted;

- and */etc/lockss.start*, which is executed at the end of */etc/rc.local*.

The implementation also uses GnuPG which (together with the libraries it needs) is accessed directly from a directory on the CD, to ensure that its testimony as to the trustworthiness of the packages to be installed can be trusted. The keys used in this process are on the write-locked floppy and are thus under the control of the appliance operator, who can add or delete keys, and also add signatures to the floppy. This is an important point; forcing our users to trust our keys would allow us to shut the system down if we chose.

The remaining part of the implementation consists of some changes to the list of files to be "crunched" together to form the command binary in the minimal system in the CD's "boot floppy".

9.1 /etc/rc.0.lockss

This script is run just before swap is turned on to ensure that there is swap space available. It examines the available hard disks and, if they have not yet been partitioned appropriately, asks permission then does so.

The first hard disk is partitioned with about a gigabyte of swap space; the remainder and all other disks are used as file systems to contain preserved content. It creates an appropriate */etc/fstab* describing them and the MFS [20] file system that will be created later in the swap space.

9.2 /etc/rc.1.lockss

This script implements the bulk of the LOCKSS "platform". We describe it in narrative form. The file name space as the script starts is defined by the */etc/fstab* written by */etc/rc.0.lockss*. It is shown in Figure 1.

It attempts to mount the CD the system was booted from and, if it cannot, calls for help. It is not unknown to find that the kernel autoconfiguration process has failed to recognize the CD from which it was booted.

It then mounts the MFS file system on */dist*. At

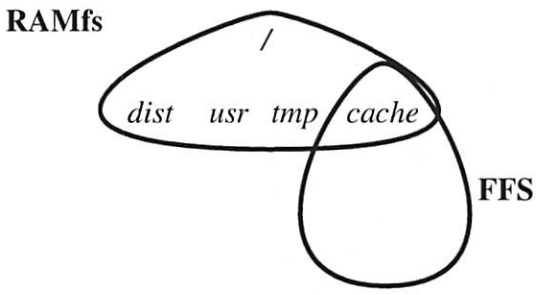

Figure 1: The file name space as */etc/rc.1.lockss* starts.

this point the */tmp* directory is still where it was on boot, in the RAMdisk, and it is about to run out of space. The script relocates */tmp* into the newly-created MFS by copying its content to */dist/tmp* and replacing it with a symlink there.

It checks to see if there is a floppy in the drive. If not, it runs the configuration process described below. If there is one but it is write-enabled, the script refuses to proceed until it is write-locked. If there is a write-locked floppy, any configuration information it contains supercedes any configuration information on the CD. The GnuPG public keyring is initialized from the floppy.

It then checks all the packages it can find on the package path and writes a script that, when later executed, will for each necessary package install the latest available version which has at least one valid signature. The package path consists of the floppy, specific directories on the CD, and the cache of downloaded packages on the hard disk. Note that during this process the entire running environment consists of the RAMdisk image from the CD, GnuPG (being run directly from the CD), and configuration data from a floppy that is known to be write-locked. The checking process is as follows:

- The network is bought up using the configuration information but without any daemons running.
- GnuPG is invoked as an unprivileged user to do a key revocation check on its keyring.
- The network is shut down.
- All detached signatures for any MD5 files found in the package path are checked. If at least one valid signature by a key that is not known to

be revoked is found, that file's list of MD5s is added to the valid-MD5 list.

- The package path is searched for versions of the necessary packages. The MD5 of each version found is computed and compared against the valid-MD5 list. If a match is found, and a lower version number of the package is already in the install-package script, it is deleted. Then a command to install this version is added to the script.
- If no unrevoked signature validates the MD5 of a package that must be installed, the boot process is aborted and the system enters "hunker-down" mode. It, and its preserved content, are inaccessible from the network and thus should be safe for some time. The administrator will need to add new signatures to the floppy before resuming normal operation.

The script next prepares for package installation by creating copies of the system directories under */dist* and replacing the originals with symlinks pointing to the copies.

The system then executes the install-package script it wrote. This first installs the OpenBSD base "packages" into directories in the MFS under */dist*, then replaces the "boot-floppy" system directories with symlinks to the installed versions in */dist*. Now the file name space approximates a properly installed OpenBSD system, and the install script continues to install a chosen set of real packages from the ports tree, and a small number of LOCKSS packages. Although any of these packages may come from any of the directories in the package path, each is known to have had its MD5 signed by at least one unrevoked key trusted by the system's administrator.

Finally, the script unmounts everything it mounted and returns to the */etc/rc* script. The file name space at this point is shown in Figure 2.

9.3 /etc/lockss.start

This script is run by */etc/rc.local* just before the end of the system boot process, after the few system daemons we have not disabled have been started. It enables the watchdog, checks for the required LOCKSS daemon configuration files and, if it finds

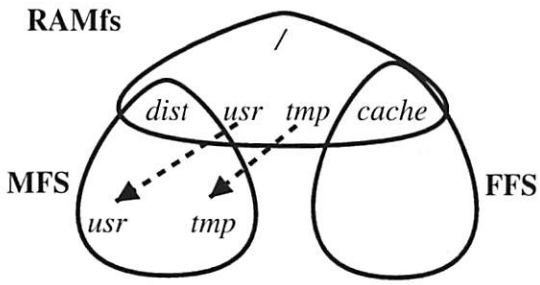

Figure 2: The file name space as */etc/rc.1.lockss* ends.

them, invokes the JVM to run the LOCKSS daemon.

9.4 Configuration

LOCKSS peers use their IP address as their long-term identity and thus, at present, do not support DHCP. If no configuration information is available, early in */etc/rc.1.lockss* a text dialog is used to acquire an IP address, netmask, gateway, DNS servers, root password etc.

The supplied values are validated by using them to bring up the network interface temporarily and, as an unprivileged user, perform some simple tests such as a DNS lookup. If the tests work, the network interface is shut down. The user is asked to write-enable the floppy, and the configuration is written to the floppy as a text file. New SSH host keys are generated and stored on the floppy, together with the initial GnuPG public keyring. The user is then asked to write-lock the floppy. The system verifies that it is write-locked before proceeding.

10 Performance

One performance issue that might make these techniques unsuitable for other applications is that it takes the system some time to boot. On a 400MHz PC, somewhat slower than those our test sites typically use, with a 40X IDE CD drive, the boot sequence takes about 320s from reset to login prompt. This includes about 30s checking signatures and hashes, about 25s installing the OpenBSD base packages, and about 180s installing about 96MB of other packages.

If our caches reboot once every 60 days this would represent about 0.006% downtime, a reasonable price to pay for low administrative costs. Although we don't have accurate data, we believe our test caches reboot significantly less often than this. Obviously, reducing the size of the packages to be installed would reduce the impact of rebooting on availability, and there is much scope for doing so.

Once the system is up, its performance is indistinguishable from a similar system running a more conventional configuration of OpenBSD.

A more critical performance measure is the time taken to respond to a security vulnerability. We simulated this by involving our test sites in a "firedrill", in which we generated and distributed a "patch" that simply sent us an e-mail confirming that the system had downloaded and was using it. The firedrill showed that building and distributing a patch is now far more efficient than it was for the Linux floppy version. With little effort on our or their part we were able to get 96% of the deployed systems upgraded within 48 hours of the start of the firedrill.

11 Deployment

In general, the design, implementation and rollout of the new version to over 50 test sites went smoothly. Almost all our test sites have been trouble-free since their upgrade in January 2003 from the floppy version. This is a great tribute to the quality of OpenBSD.

The LOCKSS appliance has survived security audits by some fairly demanding sites (e.g. CERN and LANL).

12 Problems

We did encounter a number of problems.

Our initial design involved union mounting the MFS file system over the root directory. This didn't work; it seemed to cause deadlock in the kernel. We have not verified that the problem is still present; working around it seemed preferable to diagnosing problems that early in the boot sequence.

OpenBSD appears to provide no way to check whether a floppy is write-locked or write-enabled without mounting it, and seeing whether an attempt to write to it fails. If it does fail, the kernel generates alarming error messages which the operator has to be told to ignore. OpenBSD currently lacks FreeBSD's mechanism for selectively disabling these messages.

We orignally wanted to use the system itself to burn the CDs. Again, the operator has to ignore some alarming error messages, so we put the idea aside.

The kernel autoconfiguration sometimes fails to recognize the CD from which it was booted, causing the attempt to mount the CD to fail, and preventing the system installing the software on it. Power-cycling the machine seems to be the only cure.

The native 1.3 JVM for OpenBSD wasn't available, so we run the Linux JVM under the RedHat emulator. This works well but installing the emulator and the RPM package slows the boot considerably and makes us nervous.

The RPM-based package install for the JVM insists on executing from a path in */usr*; we have to specifically turn off *noexec* during this install and re-enable it afterwards.

We ask our test sites to download an image of each new version of the CD as a *.iso* file, write a CD-R of it using some other machine, and boot their cache machine from it. The first time they tried this, about 1 in 20 sites had problems of one kind or another that required support from the team. Examples are difficulty with Windows CD-writing software, difficulty with boot device settings in the BIOS, and bad CD-R media.

Despite our suggestion that they use second-hand machines, our enthusiastic test sites often buy brand-new hardware. This frequently has motherboard Ethernet chips which are too new for the OpenBSD drivers to use. We have accumulated a reserve of really old Ethernet cards to mail out in such cases.

The security vulnerability firedrill demonstrated a need to improve our process for making a patch available. In particular, we need to focus on making our test sites aware that when we say "you need

to reboot your system right now" they need to pay attention. We hope that exercising this process regularly will make it go more smoothly when it is needed "for real"; the 96% success in 48 hours of the first drill is not good enough.

13 Related Work

Our work has similarities with KNOPPIX[11] and other bootable Linux CDs used as demo and rescue systems. They typically run directly from the CD rather than, as we do, installing system packages into an evanescent file system. We take a lot longer to boot while we do the installation, but we can validate the signatures on the packages, and thus use downloaded signed packages to upgrade the system on the CD.

Sun's Cobalt product line [13] is an excellent example of packaging Linux into an network appliance that works well with limited administration. Our automatic software upgrade mechanism was inspired by Cobalt's, which has automatic notification of availability but manual installation.

14 Future Work

Work on the platform is on hold for a while; while we focus on using it to deploy a completely re-written version of the daemon. When we get back to it, our to-do list includes in descending order of priority:

- supporting DHCP, NAT, and machines behind firewalls,
- using native Java,
- supporting USB storage devices for configuration,
- building a KNOPPIX-like LOCKSS demo mode,
- burning CDs from the system itself to allow packages and configuration data to be stored on the CD,
- revisiting the idea of union mounting MFS over the root.

There is also the possibility of extending this work to interesting and useful areas that aren't directly related to the mission of LOCKSS. Extensions to the stackable Open Source BIOS [1], which is based on LinuxBIOS [14], could provide a trusted boot sequence on which we could base our verification of the higher levels of the system. Combining this with a physically write-protected USB "dongle" containing the keys could provide almost all the capabilities of TCPA except tamper-resistance and secret key protection for Open Source systems.

15 Conclusion

In many ways, OpenBSD has proven an excellent basis for our network appliance. It was easy to adapt the install CD to our purposes. The carefully created default configuration is an excellent starting point for further restrictions. The use of privilege separation in the SSH daemon reduces the risk of allowing adminstrative access via the network. OpenBSD 3.3's addition of Stack Smashing Protection [8] adds further protection. AnonCVS access to the patch branch and the ports tree has allowed us to build from scratch every night, materially reducing the load on a small support team, especially when responding to new vulnerabilities. The OpenBSD build process is well-structured and allowed us to add a new CD image with little effort, both initially and as the base distribution evolved from 3.1 to 3.3.

We have added to OpenBSD a set of capabilities that allow it to serve quite satisfactorily as a network appliance with unskilled administrators, if only for an application that can tolerate extended reboot times. These include:

- Running the system entirely from evanescent file systems re-created from write-locked media at boot time, with no ability to execute code from a persistent file system.
- Verifying the signatures on all software during the boot process.
- Implementing a semi-automatic patch distribution mechanism for packages and their signatures.

There are, however some deficiencies in OpenBSD

that still cause significant problems as we deploy it as an appliance. Those causing the most support load are:

- The kernel produces many scary-looking error messages in non-error situations.
- The kernel does not reliably recognize low-cost IDE CD drives.
- The NIC drivers are sometimes unable to recognize or use leading-edge hardware.

There is never a perfect choice of platform for an application; all choices have advantages and disadvantages. OpenBSD has served us as well as we expected, and we hope that others implementing network appliances will find our experiences useful, whether they agree with our choice or not.

16 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 9907296, however any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

The LOCKSS program is grateful for support from the National Science Foundation, the Andrew W. Mellon Foundation, Sun Microsystems Laboratories, and Stanford Libraries.

Special thanks are due to Mark Seiden, who made major contributions to both versions of the platform and especially to the signature verification process. Thanks are due also to our long-suffering beta sites, and to the LOCKSS engineering team of Tom Robertson, Tom Lipkis, Claire Griffin, and Emil Aalto. Our web crawler is adapted from an original by James Gosling.

Vicky Reich has made the LOCKSS program possible.

The anonymous BSDcon reviewers and Todd Miller, the paper shepherd, provided many pertinent and helpful comments.

Finally, the thanks of the entire LOCKSS team go to everyone who has contributed to OpenBSD, GnuPG, the JVMs, and the Jetty web server.

17 Availability

The source for the entire LOCKSS system, including the appliance platform described above, carries BSD-style Open Source licenses and is available from the LOCKSS project at SourceForge.

References

[1] Adam Agnew, Adam Sulmicki, Ronald Minnich, and William Arbaugh. Flexibility in ROM: A Stackable Open Source BIOS. In *Proceedings of the Freenix Track: 2003 Usenix Annual Technical Conference*, San Antonio, TX, USA, June 2003.

[2] Trusted Computing Platform Alliance. TCPA. `http://www.trustedcomputing.org`.

[3] Kent Anderson, John Sack, Lisa Krauss, and Lori O'Keefe. Publishing online-only peer-reviewed biomedical literature: Three years of citation, author perception, and usage experience. *The Journal of Electronic Publishing*, 6(3), March 2001.

[4] Assoc. Research Libraries. ARL Statistics 2000-01. `http://www.arl.org/stats/arlstat/01pub/intro.html`, 2001.

[5] Andrzej Bialecki. PicoBSD. `http://people.freebsd.org/~picobsd/picobsd.html`.

[6] Mort Bay Consulting. Java http server and servlet container. `http://http://jetty.mortbay.org/jetty/`.

[7] Charles D. Cranor and Theo de Raadt. Opening The Source Repository With Anonymous CVS. In *Proceedings of the Usenix Annual Technical Conference*, Monterey, CA, USA, 1999.

[8] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/main.html`, June 2000.

[9] GnuPG. http://www.gnupg.org.

[10] Michael Keller, Victoria Reich, and Andrew Herkovic. What is a Library Anymore, Anyway? *First Monday*, 8(5), May 2003. http://www.firstmonday.org/issues/issue8_5/keller/index.html.

[11] Klaus Knopper. Building a self-contained autoconfiguring linux system on an iso9660 filesystem. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, USA, October 2000.

[12] Petros Maniatis, Mema Roussopoulos, TJ Giuli, David S. H. Rosenthal, Mary Baker, and Yanto Muliadi. Preserving Peer Replicas By Rate-Limited Sampled Voting. Technical Report cs.CR/0303026, Stanford University, March 2003. Submitted for publication.

[13] Sun Microsystems. http://www.sun.com/hardware/serverappliances/.

[14] Ronald Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.

[15] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It? In *Proc. 4th Usenix Symp.on Internet Technologies and Systems*, March 2003.

[16] Linux Router Project. http://http://www.linuxrouter.org.

[17] Niels Provos. Preventing Privilege Escalation. Technical Report 02-2, CITI, University of Michigan, August 2002.

[18] David S. H. Rosenthal and Vicky Reich. Permanent Web Publishing. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track (Freenix 2000)*, pages 129–140, San Diego, CA, USA, June 2000.

[19] David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, and Mary Baker. Economic Measures to Resist Attacks on a Peer-to-Peer Network. In *Proceedings of Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[20] Peter Snyder. tmpfs: A Virtual Memory File System. In *Proceedings of the Autumn 1990 EUUG Conference*, pages 241–248, Nice, France, 1990.

[21] Diomidis Spinellis. The Decay and Failures of Web References. *Communications of the ACM*, 46(1):71–77, January 2003.

[22] Tatu Ylonen. SSH – Secure Login Connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, USA, July 1996.

Using FreeBSD to Render Realtime Localized Audio and Video

John H. Baldwin
The Weather Channel
Atlanta, GA 30339
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

One of the largest selling points for The Weather Channel (TWC) is the ability to generate localized content for its subscribers, specifically local weather forecasts. To facilitate this localized content, TWC deploys smart devices in cable head ends. These smart devices are called STARs (Satellite Transmitter Addressable Receivers) and are responsible both for collecting weather data and displaying the data to the user in an audio and video presentation.

TWC decided to produce a next generation STAR that was more flexible than the current generation as well as cheaper. FreeBSD was chosen as the platform for these devices. Although FreeBSD was largely suitable for this application, a few modifications were required and several workarounds were employed. The end result is a PC built mostly of off-the-shelf components that is able to render broadcast quality audio and video in realtime.

1 Introduction

Most cable channel operators provide national or regional feeds from a single source to cable head end operators across the country and world. These feeds are transmitted via satellite from the broadcasters to the head ends. The head ends then distribute these feeds to viewers over cable. The Weather Channel is a unique broadcaster in that part of its presentation to viewers is localized to provide local weather observations and forecasts. It is not feasible to generate the content of these local forecasts at TWC and then distribute them over satellite to each of the head ends. There is just not enough bandwidth. Instead, a smart device known as a STAR is deployed in each head end. In addition to a national audio and video feed, TWC sends weather-related data to the STARs out in the field including local observations, thirty-six hour text forecasts, daily extended forecasts, radar images, and severe weather alerts. In addition, TWC can send software updates and other non-weather-related data over the satellite.

For example, every hour thousands of automated observation stations all over the United States sample the weather conditions at their location and transmit this data to the National Weather Service (NWS). Most STARs in the field are assigned an observation point. TWC receives this data from NWS and then sends it over the satellite where it is received by each of the STARs. Each STAR saves any data that is relevant to its configuration and discards the rest. The STAR may then display those observations either on top of the national feed in a small bar at the bottom of the screen, or it may render a full-screen presentation.

Currently TWC broadcasts its feed as an analog NTSC signal for the core network. TWC has started migrating to a digital MPEG signal to meet the demands of the cable industry as it begins the migration to digital. As part of this move, TWC is deploying a next generation STAR device known as the IntelliStar.

2 What is an IntelliStar and What Does It Do?

An IntelliStar must perform two major tasks concurrently. First, it must collect weather data from the satellite feed and store the data relevant to the location it is configured for. This data includes current observations, radar images, satellite images, and several different forecasts. Second, the device must display this weather data to viewers at broad-

cast quality that compiles with FCC regulations. This includes rendering both full screen video as well as being able to render graphic objects on top of a live video feed. When a box is down, it can not deliver local content on the analog output and can not deliver any digital output, so down time is very visible to customers and must be avoided.

In addition, since other networks do not deploy devices to head ends to generate localized content, an IntelliStar must operate as a "black box" from a cable head end technician's perspective. For example, head end technicians are accustomed to using the power button to turn things off, so the device must handle powering off via the power button gracefully. Also, the device must operate without having a keyboard, mouse, or monitor connected. Finally, the device should not require any interaction from a head end technician beyond using the power button to power the device on and off.

In addition to these requirements, TWC wants to take advantage of recent technology innovations. The current generation of STARs, known as the WeatherSTAR XL, are built from SGI boxes running IRIX with additional proprietary hardware. The IntelliStar, on the other hand, is built largely from commodity PC hardware with limited proprietary hardware. This has resulted in a cheaper and more compact device that is easier to upgrade in the future. For example, the actual rendering of video content is done using the hardware acceleration of an off-the-shelf AGP graphics adapter. Should the need for more rendering horsepower arise in the future, the graphics adapter can be replaced with a newer adapter without any changes needed in either proprietary hardware or software. To accomplish this goal, much of the realtime requirements for audio and video have been offloaded into proprietary hardware.

3 Why FreeBSD?

For the software platform on the IntelliStar, TWC also wants to stick with largely off-the-shelf software. The main device needs a stable and reliable general purpose OS that is easily maintainable. The TWC applications are written in a mixture of C++ and Python and are multithreaded, so decent support for those platforms is required. Finally, TWC needs a platform that can be easily updated in the field over the one-way satellite feed. The final decision on the OS for the IntelliStar was made in the summer of 2001. Prior to that, the development platform was Red Hat Linux 7.0. FreeBSD was chosen over Red Hat 7.0 for three primary reasons: stability and maturity of the virtual memory subsystem, a current and working tool chain, and a complete and self-consistent distribution from one source.

3.1 VM Stability

As mentioned before, down time is very noticeable. If a box is down, then thousands of viewers may not receive their local forecast. At the time of the platform decision, TWC knew that the development of the IntelliStar would take some time. Thus, TWC wanted an OS branch that was stable and mature but would not be obsolete by the time the IntelliStar was deployed. The 2.2.x Linux kernel series was nearing its end of life cycle as the 2.4.x Linux kernel series was just ramping up. However, the virtual memory subsystem of the early 2.4.x series had severe stability problems. In addition, Linux changed over to an entirely new virtual memory subsystem during the early 2.4.x series. FreeBSD, on the other hand, employed a mature, stable, and well-tested virtual memory subsystem in its 4.x branch. TWC developers noted that even on the Linux developer mailing lists, FreeBSD's virtual memory subsystem was held up as the benchmark to test the new Linux subsystems against. As a result, TWC developers were much more comfortable with FreeBSD's virtual memory subsystem than with Linux's. FreeBSD 4.x was also not nearing its end of life cycle during 2001, so TWC developers were not worried about having to perform major OS upgrades in the middle of their development cycle or immediately after the initial deployment.

3.2 Compiler

The software in TWC's STAR group largely consists of multithreaded C++ applications, and TWC required a compiler toolchain that would work with these applications. The compiler that shipped with Red Hat 7.0 was version 2.96 of the GNU C Compiler (gcc). This version of gcc was not an official release of gcc by the GNU Project. Instead, it was a snapshot of the 3.0 development branch of

GCC with additional fixes from Red Hat developers. Out of the box this pre-release compiler was unable to compile simple multithreaded C++ applications. Numerous patches were required from Red Hat before the compiler could compile this simple test case. Other Linux distributions used releases from the gcc 2.91 series which were unable to compile TWC's applications due to their more limited C++ support. FreeBSD, on the other hand, shipped with supported releases from the gcc 2.95 branch. This compiler was able to handle multithreaded C++ applications out of the box without the need for further patches and included sufficiently recent C++ support to compile TWC's applications.

3.3 Self-Contained OS

IntelliStar devices will be deployed all over the continental U.S., and TWC wanted to be able to upgrade the base OS if necessary in addition to TWC's custom applications. The only data link that is reliable for all STARs is the one-way transport from TWC out to the STARs via satellite. Thus, update systems that require a two-way link are not suitable. Since FreeBSD is an entire OS from one source rather than a collection of packages from different sources like Linux distributions, it is easier to generate an arbitrary release that is self-consistent and use that complete distribution for either installs or to upgrade units in the field. OS components can be upgraded merely by sending a tarball out to the deployed units without having to worry about keeping a local package database up to date. Formal packages are only used for third party libraries and applications that are not part of the base OS and TWC applications.

Remotely upgrading the kernel is also safer and easier with FreeBSD than with Linux. With Linux one has to ensure that the lilo boot blocks are updated after a new kernel is installed or one may end up with an unbootable box. FreeBSD's bootstrap simply looks for the loader by name in a UFS filesystem, so simply updating the actual kernel file is all that is needed to upgrade the kernel. Recent versions of Linux now offer the GNU GRUB boot loader which does not suffer from the same limitations as lilo. However, GNU GRUB is still under development and was not in use when the final OS decision was made.

Finally, TWC could easily build a custom release of FreeBSD including customized installation scripts. TWC currently maintains a small set of local patches. Using the `LOCAL_PATCHES` and `LOCAL_SCRIPTS` features of FreeBSD's release process as detailed in [release.7], TWC is able to build a full release with these patches that can be installed either from a CD or over the network like any other FreeBSD release. By using PXE network booting and FTP install over a LAN, the installation of a new machine simply involves plugging the box in and turning the power on.

4 Changes from stock FreeBSD

The current generation of IntelliStars are based on release 4.7 of FreeBSD. Although the stock FreeBSD release met many of the requirements, several modifications were required. These modifications included backporting support for the Advanced Configuration and Power Interface (ACPI) from 5.0, modifying the `sio` driver to share PCI interrupts, adjusting the maximum size of receive socket buffers, and several fixes for sysinstall to allow for more flexible scripted installations.

4.1 Backporting ACPI

As mentioned earlier, one of the requirements for the IntelliStar was that it require little interaction from head end technicians once it is deployed. Specifically, head end technicians should be able to safely power off the device by flipping the power switch. With the advent of ACPI, implementing this feature becomes possible as an ACPI system will inform an ACPI-aware OS when the power button is pressed. The OS can safely shut down and then ask the computer to power itself off via ACPI. The details of how this works can be found in the ACPI specifications which are available at [ACPI].

The 5.x development branch of FreeBSD contained a mostly functional ACPI driver including support for power button events. Thus, the same functionality could be brought to the version of FreeBSD used in the IntelliStar by backporting a stripped down ACPI driver to the 4.x branch. The majority of the ACPI driver consists of Intel's ACPI Component Architecture (ACPICA) which is OS independent and freely available at [ACPICA]. Thus the

only code that needed to be backported was the OS shim that wraps around ACPICA as well as some supporting code in other areas of the kernel used by the shim.

The changes made to the OS shim included using proper synchronization primitives and avoiding the use of a private taskqueue. 5.x uses mutexes to protect data structures while 4.x still uses the spl mechanism to protect top-half kernel code from being interrupted. Thus, macros were added that use `splhigh` and `splx` for synchronization with the ACPI interrupt handler on 4.x and a mutex on 5.x. FreeBSD 4.x also does not support the same software interrupt API used by the ACPI code in 5.x to implement a private taskqueue. Thus, the backport to 4.x simply uses the system taskqueue for ACPI events instead. These changes were wrapped in appropriate precompiler conditionals and committed to the 5.x branch prior to 5.0-RELEASE.

Changes made to other parts of the kernel include backporting entire subsystems and drivers as well as modifications to existing subsystems and drivers. The ACPI driver required support for the `bus_set_resource` and `bus_get_resource` methods in the `nexus` driver, so these changes were backported and committed to the 4.x branch prior to 4.8-RELEASE. The ACPI driver also used the `resource_list_print_type` helper function from the resource manager, so that function was backported and committed prior to 4.8-RELEASE. The ACPI driver also depended on the new power subsystem and `pmtimer` driver which debuted in 5.0-RELEASE. The actual code for both of these subsystems compiles directly on 4.x but does require some simple changes to the i386 clock and low level interrupt code. In addition, the use of the power subsystem requires several changes to the `apm` driver. Several of the changes made to the `apm` driver in the 5.x branch were merged prior to 4.8-RELEASE to minimize the size of the local patches TWC maintains.

The other significant difference between ACPI support in 5.x and ACPI support for 4.x is that the backported ACPI driver does not include the PCI support code. This means that the backported ACPI driver does not route PCI interrupts using the `_PRT` tables or enumerate host to PCI bridges. The ACPI PCI code depends on large changes to the PCI driver made in 5.x that are too large to backport to 4.x. The rest of the functionality provided by the ACPI driver is present in the backport including power button events, suspend and resume, battery status, AC adapter status, CPU throttling, thermal zones, and the ACPI timer.

4.2 Sharing `sio` PCI Interrupts

In FreeBSD, there are two main types of interrupt handlers: fast interrupt handlers and non-fast interrupt handlers. Fast interrupt handlers execute with slightly less latency to the original interrupt request and than non-fast handlers. Also, all interrupts are blocked while executing a fast interrupt handler. Fast handlers cannot share an interrupt source such as an interrupt request (IRQ) line with other interrupt handlers. Non-fast interrupt handlers, on the other hand, can share an interrupt source with other non-fast interrupt handlers. This is enforced in `bus_setup_intr` by having attempts to register a fast interrupt handler on an interrupt source that already has an interrupt handler and attempts to register a non-fast interrupt handler on an interrupt source that already has a fast interrupt handler fail.

The purpose of fast interrupt handlers is to minimize latency for devices that require very low latency. A prime example of such devices are the serial ports found in PCs which have very small data buffers. As a result, if interrupt latency is high, characters will be dropped. Thus, the driver for serial ports, `sio`, uses fast interrupt handlers.

When not using I/O APICs to manage interrupts on a PC, there are only sixteen interrupt sources in the form of ISA IRQ lines. Most of these IRQ lines are reserved for ISA devices. ISA devices cannot share interrupts, so PCI devices are restricted to using the IRQ lines not used by any ISA devices. Due to the limited number of free IRQ lines and and the increasing number of PCI devices in PCs, PCI devices are usually required to share whatever IRQ line is allocated to them with other PCI devices. Thus, if a PCI device driver uses a fast interrupt handler it can either block other devices from registering non-shared interrupt handlers on the same interrupt source if it is the first driver to register a handler for that source, or it can fail to register its handler if another driver has already registered a handler.

The `sio` driver handles the second case but does not handle the first case. To handle the second case, the

`sio` driver first attempts to register its handler as a fast interrupt handler. If that fails, it tries to register it as a non-fast interrupt handler. However, the `sio` driver currently has no way of detecting the first case and properly handling it.

The IntelliStar uses a PCI modem managed by the `sio` driver that just happens to be the first PCI device to register its interrupt handler for its interrupt source during the boot phase. As a result, other PCI devices using the same interrupt source such as an Ethernet adapter are unable to register their interrupt handler and fail to attach. The solution that TWC developed was to add a flag to the `sio` driver's global attach routine to specify whether or not the driver should attempt to register a fast interrupt handler or if it should only use a non-fast interrupt handler. The attachments for different busses can then set this flag to force `sio` to share its interrupt source with other devices. For example, the PCI and PCCard busses force the `sio` driver to share its interrupt source in TWC's patch.

The patch was submitted for review to the FreeBSD developers but was rejected as being too much of a hack. Several of the developers still wished to allow the `sio` driver to use fast interrupts when possible and wanted to fix the problem TWC had in the drivers for the busses themselves such as the PCI bus device driver instead of in `sio`. However, no progress was ever made on even how to go about doing that, so TWC continues to maintain this bug fix as a local patch.

4.3 Increasing Size of Socket Receive Buffers

As mentioned earlier, an IntelliStar receives not only live video over the satellite, but also data including weather data and software updates. For the IntelliStar, this data is transmitted over the satellite in a multicast UDP data stream alongside the video stream. The bundling of the data stream with the video stream is managed by an external integrated receiver/decoder (IRD) which provides the data stream as the original UDP multicast stream to one of the Ethernet ports on the IntelliStar. Since this communication transport is only one-way, TWC has no way of knowing if an IntelliStar has lost data due to the socket buffer overflowing. If the link were two-way, then a reliable protocol such as TCP could detect that data was lost and request a retransmission, but with the one-way link that option is not available. Since rendering the on-screen graphics is higher priority than reading data off the socket, there can be enough latency between reads of the receiving socket for the standard socket buffer size of 41600 bytes to overflow. To fix this, TWC uses `setsockopt` to increase the size of the receiving sockets buffer to 512 kilobytes. At our current data rate this allows up to one full second of data in the buffer. The default maximum size for a socket buffer in FreeBSD is 256 kilobytes, however, so TWC added an entry to `/etc/sysctl.conf` to increase this limit to four megabytes per socket buffer via the `kern.ipc.maxsockbuf` sysctl.

4.4 Scripting Enhancements for sysinstall

One of the benefits of FreeBSD mentioned in an earlier section is the ability to easily build a customized installation process. This can be done with the default installation utility, sysinstall, via installation scripts. This custom installation allows TWC to quickly and easily install all of the necessary software on IntelliStars before they are deployed into the field.

While working on the scripts for TWC's custom release, a few limitations and gaps were found in sysinstall's scripting support. TWC extended sysinstall's scripting support to address these shortcomings. All of these changes to sysinstall were committed prior to 4.7-RELEASE. To support more flexible installation scripts, TWC also made some changes to the release Makefile.

The first two changes made to sysinstall allowed scripted installs to optionally be more interactive. The first change added the `diskInteractive` variable to the disk layout editors. If this variable is set when invoking the `diskPartitionEditor` or `diskLabelEditor` commands, then the interactive disk layout editors will be used instead of requiring a fully scripted disk layout. The second change added the `netInteractive` variable to the network interface setup dialog. If this variable is set when invoking the dialog via the `tcpMenuSelect`, `mediaSetFTP`, or `mediaSetNFS` commands, then the user is asked if they wish to use DHCP or IPv6 rather than assuming that neither is desired.

The next change to sysinstall fixed a bug in the han-

dling of the `noError` variable. As documented in [sysinstall.8], the `noError` variable causes sysinstall to ignore failures from the next command executed. Usually sysinstall will abort and stop executing a script if a command fails. The `noError` variable allows a script to continue if a non-fatal error occurs. The bug was that the `noError` variable was only cleared if a command failed. Thus, if the non-fatal command immediately after `noError` was set succeeded, the variable remained set and the subsequent failure of a later command would be bogusly ignored. The fix was simply to always clear `noError` after executing a command.

The fourth change to sysinstall involved the addition of the `mediaClose` command. This command simply executes the internal function by the same name. For an install using a CD as the installation media, this will unmount the CD allowing it to be ejected from the drive. TWC uses this at the end of CD installations to unmount the CD prior to displaying a dialog box prompting the user to eject the CD.

The final changes made were to the release Makefile and not the installation utility itself. The first change was to add the dialog program to the memory filesystem used as the root filesystem during installations. This made the dialog program available to shell scripts executed by the installation scripts. This allows for more complex installation scripts that can interact with the user using the various tools described in [dialog.1].

For example, the TWC install begins with a menu box prompting the user for the type of machine to install: an IntelliStar or a development machine. Depending on which option the user selects, different parameters are used. This is accomplished by having the top level install script execute a shell script. This shell script uses the dialog command to display the menu and obtain the user's choice. The shell script then generates a configuration script on the fly. After the shell script finishes, the top level configuration script loads the configuration script generated by the shell script and executes it.

To support this change, TWC added a `NO_FLOPPIES` variable to the release Makefile to disable building of boot floppies. Adding the dialog program to the memory filesystem made the memory filesystem too large to fit on floppies. By defining the `NO_FLOPPIES` variable during the release build, TWC's custom release completed without an error. TWC does not use floppies for any of its installations, so the loss of floppies as a installation boot media was not a problem. The addition of the `NO_FLOPPIES` variable was committed prior to 5.0-RELEASE and will be merged to the 4.x branch prior to 4.9-RELEASE.

5 Workarounds for FreeBSD Problems

In addition to the problems above, FreeBSD is lacking in two other areas as well that TWC chose to work around in its own software instead of modifying FreeBSD. The first area involves negative nice priorities and is worked around fairly easily. The second area consists of a couple of problems with the userland thread implementation employed in FreeBSD 4.7-RELEASE.

5.1 Nice is too Unnice

The TWC software that runs on the IntelliStar consists of several applications of varying importance. For example, rendering the video presentation is very important and receiving data is slightly less important. Most other tasks are not all that important as far as latency is concerned. Thus, negative nice values are applied to the important applications to ensure that they are not starved by any background processes.

Initially a nice value of -20 was used for the most important process. However, during development an infinite loop bug was encountered and the box locked up. Some simple tests of a program that executed an infinite loop at a nice value of -20 verified that the looping process starved all other user processes on the box. This was surprising since it was expected that the CPU decay algorithm of the scheduler would sufficiently impact the priority of the important process so that other userland processes would receive some CPU time. As a matter of fact, the CPU decay algorithm will not decay a nice -20 process enough to allow normal processes with a nice value of zero to execute. The explanation can be found in a simple examination of the code.

The priority of a userland process is calculated by the following code snippet from the `resetpriority`

function:

```
newpriority = PUSER +
    p->p_estcpu / INVERSE_ESTCPU_WEIGHT +
    NICE_WEIGHT * p->p_nice;
newpriority = min(newpriority, MAXPRI);
p->p_usrpri = newpriority;
```

The p_nice member of struct proc holds the nice value and p_estcpu holds an estimate of the amount of CPU that the process has used recently. This field is incremented every statclock tick in the schedclock function:

```
p->p_estcpu = ESTCPULIM(p->p_estcpu + 1);
```

The ESTCPULIM macro limits the maximum value of p_estcpu. Its definition along with the definition of other related macros follows:

```
#define NQS     32
#define ESTCPULIM(e) \
    min((e), INVERSE_ESTCPU_WEIGHT * \
        (NICE_WEIGHT * PRIO_MAX - PPQ) + \
        INVERSE_ESTCPU_WEIGHT - 1)
#define INVERSE_ESTCPU_WEIGHT   8
#define NICE_WEIGHT     2
#define PPQ             (128 / NQS)
#define PRIO_MAX        20
```

Thus, the maximum value of p_estcpu is 295.

Since p_estcpu is never less than zero, a process with a nice value of zero will have a userland priority greater than or equal to PZERO. For a process with a nice value of -20, the total nice weight ends up being -40. However, the maximum weight of the CPU decay is 36. Thus, with a nice value of -20, the CPU decay algorithm will never overcome the nice weight. Thus, a lone nice -20 process in an infinite loop will starve normal userland processes with a nice value of zero. In fact, since the maximum CPU decay is 36, any nice value less than -18 will produce the same result.

However, according to a comment above updatepri, p_estcpu is supposed to be limited to a maximum of 255:

```
/*
 * Recalculate the priority of a process after
 * it has slept for a while.  For all load
 * averages >= 1 and max p_estcpu of 255,
 * sleeping for at least six times the
 * loadfactor will decay p_estcpu to zero.
 */
```

If this is the case, then the maximum CPU decay weight is merely 31, and any nice value less than -15 can starve normal userland processes.

One possible solution would be to adjust the scheduler parameters so that a nice -20 process did not starve userland processes. For example, INVERSE_ESTCPU_WEIGHT could be lowered from eight to four. However, increasing the strength of the CPU decay factor in the scheduling algorithm might introduce other undesirable side effects. Also, such a change would require TWC to maintain another local patch to the kernel. TWC decided to keep it simple and stick to nice values of -15 and higher.

5.2 Userland Threads

Two of the larger problems TWC encountered were due to limitations in FreeBSD's userland threads implementation. As mentioned earlier, TWC's software consists largely of multithreaded C++ applications. Both problems stem from all userland threads in a process in FreeBSD sharing a single kernel context. First, when one thread calls a system call that runs in the kernel, all of the threads in that process are blocked until the system call returns. Secondly, all the threads in a process are scheduled within the same global priority. Both of these problems are demonstrated in one of the TWC applications that contains two threads. One thread is responsible for rendering frames, and the other thread loads textures into memory from files. The rendering thread is much more important than the loading thread since the loading thread preloads textures and can tolerate some latency whereas the rendering thread must pump out at least thirty frames every second.

When the loading thread is loading a large file into memory, it can temporarily starve the rendering thread. The internal implementation of the read function in the thread implementation uses a loop of non-blocking read system calls. However, if the entire file is resident in memory already, then the non-blocking read will copy out all of the file to

userland. Especially for large files, this data copy may take up enough time to delay the rendering thread by a few frames. To minimize the effects of this long delay, reads of large files are broken up into loops that read in files four kilobytes at a time. After each read, `pthread_yield` is called to allow the rendering thread to run. If the two threads did not share their kernel context, then when the rendering thread is ready to run it could begin execution on another CPU immediately rather than having to wait for the copy operation to complete.

The second problem is that all threads within a process share the same global priority. In the application in question, the rendering thread is the most important user thread in the system. Therefore, its process has the highest priority. The loading thread, however, is less important than threads in some of the other processes executing TWC applications. Since the two threads share the same global priority, the loading thread ends up with a higher priority than the more important threads in other processes. If the two threads had separate kernel contexts, then the rendering thread could keep its high priority without requiring the loading thread to have a higher priority than threads in other processes. TWC currently does not employ a workaround for this problem and so far no real world anomalies have been attributed to it.

TWC considered using an alternate thread library to work around these problems. Specifically, the thread library contained in the LinuxThreads port described at [LinuxThreads]. However, there are binary incompatibilities between the structures defined by FreeBSD's thread library and the LinuxThreads' thread library. Thus, any libraries used by a multithreaded application that use threads internally must be linked against the same thread library as the application. As a result, for our applications to use LinuxThreads, all of the libraries they link against that use threads internally would also have to link against LinuxThreads. As a result of those libraries using LinuxThreads, other applications that use those libraries would also have to link against LinuxThreads. This would require TWC to custom compile several packages including XFree86, Mesa, Python, and a CORBA ORB as well as other applications depending on those packages rather than using the pre-built packages from stock FreeBSD releases. Since the workarounds for FreeBSD's thread library were not too egregious, they were chosen as the lesser of two evils.

Looking to the future, TWC is very excited about the ongoing thread development in FreeBSD's 5.x series. The more flexible threading libraries in that branch should eliminate most of the current problems with FreeBSD's current thread library. At the moment, however, TWC is uncomfortable with deploying 5.x until it is more proven and mature.

6 Conclusion

While FreeBSD may not have been a perfect fit out of the box for the IntelliStar, it was successfully adapted to the IntelliStar's needs with relatively minor effort. The first IntelliStar units began generating and delivering content to live viewers in March of 2003. As of the time of this writing 24 units are deployed across the continental U.S. All of these units deliver the Weatherscan Local network which delivers 24/7 localized weather programming. More Weatherscan units are schedule to roll out during the rest of the year, and IntelliStars should begin replacing older STARs on the main TWC channel in 2004.

7 Acknowledgments

Thanks to The Weather Channel for funding the writing of this paper, the FreeBSD development described in this paper, and numerous other FreeBSD improvements. Thanks also to all the of the contributors to the FreeBSD Project. Without their efforts, there would not be an OS to build upon. Special thanks to those who reviewed and critiqued this paper including Chris McClellen, Sam Leffler, and Gregory Shapiro. A special thanks is due as well to the FreeBSD ACPI developers who added the initial ACPI support to FreeBSD. Without that contribution, implementing soft power-off would have been much more difficult and very likely much more of a hack.

8 Availability

All of the changes to FreeBSD that have not already been committed to the tree are freely available at

the following URL:

`http://www.FreeBSD.org/~jhb/patches/`

Specifically, there are three patches of interest in that directory. The first two patches are for the backport of ACPI to 4.7-RELEASE and 4.8-RELEASE and are contained in the files `acpi_4.7.patch` and `acpi_4.8.patch`, respectively. In addition to those patches, the following files and directories must be checked out from the 5.x branch on top of an appropriate kernel source tree:

- `sys/contrib/dev/acpica`
- `sys/dev/acpica`
- `sys/i386/acpica`
- `sys/i386/include/acpica_machdep.h`
- `sys/i386/isa/pmtimer.c`
- `sys/kern/subr_power.c`
- `sys/sys/power.h`

The other patch is in the file `sio_shareirq.patch` and includes the changes to fix the `sio` driver to share interrupts with other PCI devices.

More information about FreeBSD and the FreeBSD Project is available at [FreeBSD]. For more information about The Weather Channel and its products, please see [TWC].

References

[ACPI] ACPI - Advanced Configuration and Power Interface, `http://www.acpi.info`

[ACPICA] Instantly Available Technology - ACPI, `http://developer.intel.com/ \ technology/iapc/acpi/downloads.htm`

[dialog.1] *Dialog*, FreeBSD General Commands Manual, `http://www.FreeBSD.org/cgi/ \ man.cgi?query=dialog&sektion=1& \ manpath=FreeBSD+4.8-RELEASE`

[FreeBSD] FreeBSD Project, `http://www.FreeBSD.org`

[GNU GRUB] GNU GRUB, `http://www.gnu.org/software/grub`

[LinuxThreads] LinuxThreads, `http://www.freebsd.org/cgi/ \ url.cgi?ports/devel/linuxthreads/ \ pkg-descr`

[NWS] NOAA - National Weather Service, `http://www.nws.noaa.gov`

[Python] Python Programming Language, `http://www.python.org`

[Red Hat] Red Hat, `http://www.redhat.com`

[release.7] *Release*, FreeBSD Miscellaneous Information Manual, `http://www.FreeBSD.org/cgi/ \ man.cgi?query=release&sektion=7& \ manpath=FreeBSD+4.8-RELEASE`

[sysinstall.8] *Sysinstall*, FreeBSD System Manager's Manual, `http://www.FreeBSD.org/cgi/ \ man.cgi?query=sysinstall&sektion=8& \ manpath=FreeBSD+4.8-RELEASE`

[TWC] The Weather Channel, `http://www.weather.com/aboutus`

[XFree86] XFree86, `http://www.xfree86.org`

Tagging Data In The Network Stack: *mbuf_tags*

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

We describe the *mbuf_tag* API, a mechanism for tagging data as they flow through the network stack. Originally introduced in OpenBSD, *mbuf_tags* were initially intended for use by the IPsec stack. The API has matured enough to be used by several other kernel components, and formed the basis for the FreeBSD *mbuf_tags*. We present the API, discuss its various uses in the OpenBSD network stack, and describe some plans for future work. Our goal is to demonstrate the flexibility of this relatively simple mechanism and expose it to other kernel developers.

1 Introduction

The OpenBSD [1] *mbuf_tags* framework allows the kernel to attach arbitrary information to data packets as they flow through the network stack. This information is generally of two types: *(i)* a record of processing that has already been applied to the packet, *e.g.,* the fact that a packet was encrypted under a particular IPsec [5] security association, or *(ii)* a "reminder" to perform some operation to the packet in the future, *e.g.,* apply an encryption algorithm in software prior to transmitting the packet, if the outgoing interface does not provide integrated cryptographic facilities.

In the former case, the information is intended for consumption by the kernel itself (*e.g.,* detecting whether a processing loop has occurred, to avoid resource exhaustion) or by a user-level process (*e.g.,* exposing to a network daemon some IPsec-related information, indicating that a packet was protected by a particular security association). In the latter case ("reminders"), the information is intended for use by the lower levels of the network stack, *e.g.,* device drivers that offer specific functionality, such as outgoing packet checksumming.

Although originally developed for use by the IPsec stack, *mbuf_tags* have been in use by several other network components, such as various pseudo-devices, the packet filtering (PF) engine, and some device drivers. The use of *mbuf_tags* by such diverse elements demonstrates their effectiveness and usefulness as tools for the kernel developer. This is underlined by their adoption in the FreeBSD kernel for use in the recently revised IPsec stack.

The purpose of this paper is to expose the *mbuf_tags* mechanism to the general kernel developer community, both to encourage wider use and to solicit improvements to its functionality. We believe that *mbuf_tags* offer a flexible and simple mechanism that enables several types of processing that were previously difficult or impossible to perform in the BSD network stack.

The remainder of this paper is organized as follows. Section 2 discusses the design rationale and presents the API itself in some detail. Section 3 discusses the various uses of the *mbuf_tags* in the OpenBSD network stack, and Section 4 discusses some of our future work plans. Section 5 concludes the paper.

2 The OpenBSD *mbuf_tags*

In this section, we describe the design rationale and the API for *mbuf_tags*.

2.1 Design Rationale

The *mbuf_tags* were originally developed for use in conjunction with the OpenBSD IPsec stack [6]. Their primary purpose, described in more detail in Section 3, was to record how "securely" each packet was received, *i.e.,* under what IPsec security association(s) was a packet received. Practically immediately, however, a second use was established: to detect loops in outgoing IPsec packets. For this, we needed to record the same information as in the incoming packet case. However, we (correctly, as it turns out) foresaw the need for adding tags with different types of information in the future. Thus, we chose an approach similar in some respects to the BSD *struct sockaddr* for recording network address information.

```
struct m_tag {
        SLIST_ENTRY(m_tag) m_tag_link;
        u_int16_t          m_tag_id;
        u_int16_t          m_tag_len;
};

struct m_tag *m_tag_get(int type, int len, int flags);
struct m_tag *m_tag_find(struct mbuf *m, int type, struct m_tag *tag);
struct m_tag *m_tag_first(struct mbuf *m);
struct m_tag *m_tag_next(struct mbuf *m, struct m_tag *tag);
struct m_tag *m_tag_copy(struct m_tag *tag);

void          m_tag_free(struct m_tag *tag);
void          m_tag_prepend(struct mbuf *m, struct m_tag *tag);
void          m_tag_unlink(struct mbuf *m, struct m_tag *tag);
void          m_tag_delete(struct mbuf *m, struct m_tag *tag);
void          m_tag_delete_chain(struct mbuf *m, struct m_tag *tag);
void          m_tag_init(struct mbuf *m);

int           m_tag_copy_chain(struct mbuf *from, struct mbuf *to);
```

Figure 1: **The *mbuf_tags* API.**

mbuf_tags consist of two parts: a fixed-size header, which contains the length and type of the tag as well as a pointer to other tags attached to the same packet, followed by type-dependent data. These tags can be combined together in a chain, and attached to the first *mbuf* of a packet. An *mbuf* is a data structured used by the BSD networking stack to contain packets and other information. A chain of *mbufs* can be used to store large packets, with the first *mbuf* containing additional information about the packet as a whole.

These tags can serve multiple roles, as we shall see in Section 3. They can indicate processing that has already occurred to the packet (*e.g.*, IPsec SA under which a packet was received), or it may represent a "reminder" for processing that must be applied to the packet in the future (*e.g.*, cryptographic processing that must be done to the packet by a combined network+cryptographic accelerator card).

A similar approach was taken by NetBSD, in the form of *aux mbufs*. These are *mbufs* that are attached to an *mbuf* header in a way similar to *mbuf_tags* chains. Because they use unmodified *mbufs*, the former enable the use of all the *mbuf*-manipulating routines and, perhaps more importantly, allow space to be allocated in chunks without resorting to the kernel memory allocator with every allocation, as is the case with the use of *malloc(9)* in the *mbuf_tag* approach. Thus, the processing cost of adding new tags to a packet is a step function with *aux mbufs*, whereas it increases linearly with the number of tags attached to a packet in our scheme. However, the *mbuf_tag* approach does not place more pressure on the *mbuf* allocator, which can run out of space on a busy router or firewall, since *mbufs* are allocated from a reserved area of memory, which is typically fixed to a certain percentage of kernel memory at kernel-configuration time. Furthermore, we intend to use memory pool, as we discuss in Section 4, to reduce to overhead of the kernel memory allocator. Finally, *mbuf_tags* are better-integrated with the *mbuf* subsystem, allowing for seamless replication and de-allocation when the respective *mbufs* are duplicated or released.

Linux uses a 48-byte array in the *skbuff* structure, which the "owner" of a packet (the protocol or socket that queued the packet) can use to store private information. Apart from its limited size, the ownership semantics of this array make it unsuitable for use in certain scenarios (*e.g.*, when the producer and the consumer of a tag are separated by code that performs its own processing that requires use of the array).

FreeBSD recently adopted *mbuf_tags*, adding a *cookie* field in the tag header. This allows for private, module-specific definition of new tags without requiring coordinated allocation among different modules/developers. We intend to include this change in the OpenBSD tag implementation.

2.2 Tags API

The *mbuf_tag* API is shown in Figure 1. The code implementing the API is contained in the file

sys/kern/uipc_mbuf2.c of the OpenBSD distribution.

m_tag_get() allocates a new tag of type type with *len* bytes of space following the tag header itself. The flag argument is passed directly to the kernel *malloc(9)*. If successful, *m_tag_get()* returns a memory buffer of *(len + sizeof (struct m_tag))* bytes. The first *sizeof (struct m_tag)* bytes will contain a tag header, the definition of which is also given in Figure 1. The first field contains a pointer to other tags on the same mbuf. The length field contains the size, in bytes, of the array following the tag header itself. There are several types defined, and we describe their use in Section 3. *m_tag_free()* de-allocates a tag.

m_tag_find() finds an instance of a tag of the given type. The caller can specify that the search start from an arbitrary point in the tag list (as indicated by the third argument). This allows the caller to examine all tags of a given type that are attached to a packet, by repeatedly calling *m_tag_find()*, as shown in Figure 2.

```
/*
 * This code can be written
 * better, but this fits in
 * the two-column format :-)
 */
tag = m_tag_find(m, type, NULL);
while (tag != NULL) {
   ... code examining the tag ...
   tag = m_tag_find(m, type, tag);
}
```

Figure 2: **Using** *m_tag_find()*.

For clarity, the *m_tag_first()* and *m_tag_next()* pair of calls can be used to the same effect.

m_tag_prepend() links a new tag to the head of the list. Tags are typically attached in a manner that reflects the order in which the operations they represent were applied to the packet. For example, a packet that is processed (*e.g.*, decrypted) by IPsec twice will have two attached tags, the first of which (as returned by *m_tag_find()*) will represent the second decryption. *m_tag_unlink()* detaches a tag from the packet, without deallocating the memory. *m_tag_delete()* combines *m_tag_unlink()* and *m_tag_free()*. *m_tag_delete_chain()* unlinks and frees all tags attached to a packet, starting from a caller-specified tag. If the last argument is left *NULL*, all attached flags will be deleted.

m_tag_copy() creates an identical copy of a tag. *m_tag_copy_chain()* creates a copy of the tag list attached to a packet and attaches it to another packet.

Finally, *m_tag_init()* is called by kernel components that manually initialize *mbufs*. There are only a handful of such locations, practically all of them in device drivers that perform their own buffer management (*e.g.*, maintaining a cache of *mbufs*).

3 Using *mbuf_tags*

We now describe the various uses of the tags in the OpenBSD network stack. The tags currently in use can be classified into four categories: IPsec, loop-detection, PF (packet filter), and miscellaneous. We describe each of these categories in turn.

3.1 IPsec-related Tags

This category includes tags that are used internally by the IPsec stack [6] to detect processing loops and propagate information to high-level protocols.

- IPSEC_IN_DONE records the fact that the packet was received under a particular IPsec Security Association (SA). If the packet has been encrypted under several SAs, there will be one such tag for each SA, with the most recently processed SA located closer to the head of the list. The tag contains enough information to locate the SA data structure in the relevant kernel database. This is used for two purposes:

  1. Determine whether a packet has been processed by an SA that satisfied the IPsec policy requirements, *e.g.*, "all TCP packets from host A must arrive encrypted". There are various locations in the network stack where such checks are performed.
  2. Propagate the information to the socket layer, whereby it is made available to applications via the *getsockopt()* call. Thus, applications can determine whether a connection is protected, the relevant parameters, the peer's identity (*e.g.*, public-key certificate), *etc.*

  This is one of the few tags that is used both inside and outside the kernel component where it is created (IPsec stack). The fact that tag processing (in particular freeing) is integrated with *mbuf* processing (freeing) helped in limiting the amount of supporting code that needed to be added throughout the stack.

- IPSEC_OUT_DONE records IPsec SAs that have been applied to an outgoing packet. This is primarily used to catch processing loops in the network stack, which could cause repeated processing (encryption) of a packet under the same set of SAs. This is necessary because the IPsec standards [5] require support for nested SA processing. Consider the following legitimate policy: "all packets to subnet 10.1.2.0/24 must be encrypted to the security gateway 10.1.2.1". Notice that the security gateway's address lies within the destination subnet's address space. A packet that matched this rule once would thus repeatedly match it every time it was re-evaluated by the IPsec policy database, causing a loop. Using the IPSEC_OUT_DONE tag, we can detect this cycle (or any cycle, of arbitrary length) and transmit the packet without further IPsec processing.

- IPSEC_IN_CRYPTO_DONE is issued by device drivers to indicate that an incoming IPsec packet has been successfully processed by a network card that has integrated support for IPsec, indicating the SA(s) processed. Incoming packets undergo regular IPsec processing; just prior to decryption/verification, the kernel checks for the presence of this tag for the specific SA. If this is present, decryption is skipped and processing continues as if it were successful. This allowed us to integrate IPsec-offloading support with less than 10 lines of kernel code.

- IPSEC_OUT_CRYPTO_NEEDED is used for the outgoing case of using a network card with integrated cryptographic processing. If the kernel is aware that the outgoing network interface offers such capabilities it simply attaches this tag to the packet, again indicating which SA it should be processed under. The device driver is then responsible for loading the SA parameters to the network card (if necessary), and for indicating to the hardware that IPsec processing under that SA is needed.

- IPSEC_IN_COULD_DO_CRYPTO is issued by device drivers that detect incoming IPsec packets for which they do not have the SA. The IPsec stack can use this tag as a signal that cryptographic processing can be off-loaded to the network interface. Although the device driver could silently load the relevant SA for end-systems, the situation is more complicated for gateways and firewalls that allow IPsec traffic to traverse them: in that case, the driver may not know which SAs are "local" and which refer to hosts behind the firewall. Since such knowledge is implicitly available to the network stack (different code will be executed), we made that code responsible for SA loading.

  Furthermore, the IPsec stack is (or can be) more aware about usage patterns across multiple SAs and can make better-informed decisions as to how best to use the limited resources available in the network card (such cards can typically support a limited number of SAs in their internal RAM).

- IPSEC_PENDING_TDB is used by the network stack to indicate that IPsec processing should occur to the packet before it is transmitted to the network. One tag for each SA that needs to be applied to the packet is attached, in the order in which they must be applied. This tag is necessary because of the requirement for SA-bundle processing (*i.e.*, policy may require that a packet be processed by a series, or "bundle", or SAs — not just one SA) and the fact that in OpenBSD cryptographic processing uses continuations [8].

3.2 Loop-Detection Tags

These tags are issued and consumed entirely by the bridge [7], *gif* interface, and *gre* interface subsystems respectively. Their main purpose is to detect processing cycles that would cause endless encapsulation or layer-2 packet forwarding. In all cases, the packet is dropped (in contrast to the IPsec loop-detection recovery, discussed in the previous section).

Systems without *mbuf_tags* have addressed the problem of loop detection/avoidance through ad-hoc and unsafe methods. In most cases, processing is assumed to be single-threaded; loops are detected through the use of locks or global variables. Not only is this approach is infeasible in multi-threaded and SMP kernels, it will also not work in certain configurations. For example, consider the case of two or more virtual bridges sharing two Ethernet interfaces: a packet scheduled for transmission in one interface of one bridge will be also scheduled for transmission on the other interface of the same bridge, also "jumping" to the second virtual bridge, whereupon it will be scheduled for transmission on the original Ethernet interface, whereupon the cycle will restart. The previous method cannot detect this failure, because bridge processing occurs at a software interrupt, making it impossible to keep packet state independently of the packet. Other similar cases arise then two or more of GRE, IP-in-IP, and bridging are combined in particular ways.

Furthermore, use of locks or global variables for detecting re-entry would make it difficult to implement

some legitimate configurations, *e.g.,* hierarchical bridges (whereby traffic propagated through one set of interfaces is also sent through a second set, but not vice versa). Admittedly, these uses are rather arcane and error-prone — they are mentioned only as an example of the flexibility of using tags for loop detection.

- BRIDGE is used by the bridge subsystem [7] to detect loops. The tag contains a pointer to the bridge interface that already forwarded the frame, allowing multi-bridge packet processing.
- GIF is used by *gif*, a network pseudo-interface that implements IP-in-IP encapsulation [9], to detect loops. Such loops are possible when the outer IP header, which is attached during *gif* processing, has a destination address that will cause the routing table to transmit the packet through the same *gif* interface again. The tag contains a pointer to the *gif* interface that processed the packet.
- Finally, GRE is used by the GRE encapsulation [3] code to detect cycles. The details are the same as with the GIF case; here, IP packets are encapsulated within GRE (and then IP) frames, and the tag contains a pointer to the *gre* interface that processed the packet.

3.3 PF-related Tags

These tags are used exclusively by PF, the OpenBSD packet filtering engine [4]. Unless indicated otherwise, these tags do not carry any additional data.

- PF_GENERATED is used to mark packets that are *generated* by PF itself, *e.g.,* ICMP messages indicating a dropped packet, or firewall-generated TCP RST packets. Such packets should not be subjected to the PF filtering rules, thus PF unconditionally accepts packets that carry this tag.
- PF_ROUTED is used to mark packets that are *routed* by the packet filtering engine, *e.g.,* using the *rdr* rule. Such packets are not tested by PF more than once, to prevent loops caused by subsequent matching routing rules.
- PF_FRAGCACHE is used to mark fragmented packets cached by PF. PF may cache such fragments as directed by its configuration, for traffic normalization purposes, *e.g.,* to avoid overlapping-fragment attacks. Packets with this tag have been cached by the fragment cache already and will short-circuit it if processed again. If they were to re-enter the fragment cache, they would be indistinguishable from a duplicate packet, and would be dropped.
- PF_QID is used by PF to indicate to the network traffic-shaping discipline, ALTQ, which queue the packet should go to. The tag contains the identifier of the queue.
- PF_TAG is used by PF to tag packets with user-defined information, and filter on those later on. Effectively, the tag is an internal marker that can be used to identify these packets. For example, such tags can be used to propagate information between input and output filtering rules on different interfaces, or to determine if packets have been processed by address-translation rules. These tags are *sticky*, meaning that the packet will be tagged even if the rule that attaches the tag is not the last matching rule. Further matching PF rules can replace that tag with a new one, but will not remove a previously-applied tag. A packet is only ever assigned one tag at a time.

3.4 Miscellaneous Tags

- IN_PACKET_CHECKSUM is used by network cards that can compute complete packet checksums to pass that information to higher-level protocols. That tag contains the 2-byte checksum of the complete packet. A protocol such as TCP needs to "subtract" the non-relevant parts of the packet from the checksum. This type of support was added for some of the older Intel cards, that did not compute protocol-specific checksums as newer hardware does.

4 Future Directions

There are several improvements we intend to make to the current API. More specifically:

- Use of memory pools (see the *pool(9)* manual page) and tag-specific deallocation routines, to improve performance. The limitation of using memory pools is that it supports fixed-size allocations which can lead to either inefficient use of memory or to a large numbers of pools. Fortunately, it appears that all the tags that have been defined to date fall in one of three categories with respect to memory allocation: they require no additional memory (beyond the tag header itself) — as was the case with

most of the PF tags, see Section 3.3, one extra word (the loop-detection tags, Section 3.2), or an IPsec SA identification payload (Section 3.1). Thus, we could simply have three different memory pools, one for each size. This change is fairly simple and does not require any changes in the API itself, so we intend to integrate it fairly soon.

- Tag-triggers, which will invoke specific packet-processing at various points in the network stack, depending on existing tags. One example is calculating the TCP or IP checksum of a packet that is about to be encapsulated inside another protocol. In the extreme case, these tags will carry a pointer to a protocol-specific function and enough data to indicate the location where the desired operation should take place. This is particularly useful when used in conjunction with network cards that support some type of functionality offloading and IPsec: if only some packets are IPsec-processed, we need a way to defer expensive processing (such as checksum computations) as late as possible, under the assumption that the packet will not be encrypted and thus the expensive operation can be offloaded to the NIC. When that assumption is violated (*i.e.,* the packet does need to be encrypted), we need to detect and apply the deferred operation. Such deferred processing is already done for TCP and UDP checksum computation, but the approach is tailored for that application. We intend to create a more general framework for deferred processing using *mbuf_tags*.

- An API for application-defined tags. This will be used either directly by applications or through *setsockopt()* calls to attach information to packets that will cause deferred processing. We have an application of this, for accelerating TLS [2] and SSH in the presence of network cards with integrated cryptographic functionality. In that scenario, the tags are used by crypto-aware network interfaces to provide application-layer protocol encryption. We intend to investigate this approach further in future work. Naturally, the type of tags that can be attached by applications must be carefully controlled, since these tags can affect network processing in ways that may not have been intended or allowed by the system administrator (*e.g.,* bypassing PF rules). Fortunately, doing so should be fairly straightforward, since the relevant interface to the kernel (*setsockopt()*) is "narrow" enough that we can perform the necessary checks.

5 Conclusions

We have presented the OpenBSD *mbuf_tags*, a mechanism for tagging packets as they flow through the network stack. These tags are used by many different kernel components such as the IPsec stack, various pseudo-interfaces, the packet filtering engine (PF), *etc.* We discussed the design rationale, the API, and the uses of the tags in OpenBSD, as well as some future improvements we intend to make. The *mbuf_tags* have been in use in OpenBSD for several years, and were recently ported to FreeBSD. *mbuf_tags* represent a powerful and flexible mechanism for allowing kernel developers to perform certain types of processing on packets in different parts of the network stack.

References

[1] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.

[2] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Comments (Proposed Standard) 2246, January 1999.

[3] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic routing encapsulation (GRE). Request for Comments 2784, Internet Engineering Task Force, March 2000.

[4] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 171–180, June 2002.

[5] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[6] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom)*, pages 1948–1952, November 1997.

[7] A. D. Keromytis and J. L. Wright. Transparent Network Security Policy Enforcement. In *Proceedings of the USENIX Technical Conference*, pages 201–214, June 2000.

[8] A. D. Keromytis, J. L. Wright, and T. de Raadt. The design of the openbsd cryptographic framework. In *Proceedings of the USENIX Technical Conference*, pages 181–196, June 2003.

[9] C. Perkins. IP encapsulation within IP. Request for Comments 2003, Internet Engineering Task Force, October 1996.

Fast IPSec: A High-Performance IPsec Implementation

Samuel J. Leffler

Errno Consulting
sam@errno.com

ABSTRACT

Fast IPsec is an implementation of the IPsec protocols [Kent & Atkinson, 1998a] for FreeBSD that was designed for high performance. In particular the protocols use the OpenBSD Cryptographic Framework, as ported to FreeBSD [Leffler, 2003], so any cryptographic hardware is automatically used to accelerate their operation. Fast IPsec, running on a uniprocessor system with a single Broadcom BCM5822 cryptographic processor, has demonstrated throughput of more than 400 megabits/second when acting as an IPsec terminator. This is more than 50% higher than any other freely available IPsec implementation.

1. Background and Introduction

The IP Security protocols (**IPsec**) are a suite of protocols [Kent & Atkinson, 1998a] standardized by the IETF [Kent, 1998; Kent & Atkinson, 1998b] for secure communication of IP datagrams. IPsec is comprised of three protocols: AH, ESP, and IPCOMP. AH provides authentication, ESP encryption and optionally authentication, and IPCOMP adds compression. Any or all of these protocols can be combined though most uses of IPsec employ only the ESP protocol (with the optional authentication). Several freely available implementations of the IPsec protocols exist: the KAME Project distributes an IPsec suite that has been integrated into FreeBSD and NetBSD, OpenBSD has their own IPsec implementation, while the FreeS/WAN Project distributes an IPsec implementation for Linux. Of these implementations only the OpenBSD software includes support for using cryptographic hardware to accelerate the protocols. Cryptographic hardware vendors sometimes provide software to integrate their products with some of these IPsec implementations but these tend to be special-purpose and only a few are freely available [Communications, 1999].

Fast IPsec is an implementation of the IPsec protocols for FreeBSD that was designed for high performance. In particular the protocols use the OpenBSD Cryptographic Framework, as ported to FreeBSD, so any cryptographic hardware is automatically used to accelerate their operation. Fast IPsec, running on a uniprocessor system with a single Broadcom BCM5822 cryptographic processor, has demonstrated throughput of more than 400 megabits/second when acting as an IPsec terminator [Ambrisko, 2003]. This is more than 50% higher than any other freely available IPsec implementation.

Fast IPsec is derived from the KAME IPsec, but integrates many of the lessons OpenBSD learned when adding hardware acceleration to their IPsec implementation. The decision to start from the KAME implementation was made for two reasons:

1) The KAME distribution is the most widely used IPsec implementation for BSD systems. Providing a successor implementation that is familiar to existing users simplifies adoption.
2) The OpenBSD IPsec is tightly integrated with other facilities that conflict with or duplicate existing facilities in FreeBSD.

While Fast IPsec has been designed as a successor to the KAME IPsec it has also been written to coexist with the KAME implementation in the source tree. This permits users to evaluate Fast IPsec without giving up their existing IPsec implementation. Further, because Fast IPsec and KAME share APIs, users do not need to learn a new set of configuration tools.

The remainder of this paper is organized as follows. Section 2 describes the Fast IPsec implementation. Section 3 discusses performance issues and the techniques used to attain its high performance. Section 4 describes performance results for FreeBSD and compares them to other freely available IPsec implementations. Section 5 outlines the the status and availability of this work and talks about future work. Section 6

gives conclusions.

2. Fast IPsec Implementation

Fast IPsec is comprised of four protocols (AH, ESP, IPIP, and IPCOMP), the security database, **PF_KEY** socket support through which applications interact with the database, and glue code that resides in protocols such as IP, TCP, UDP, and ICMP. There is no cryptographic transformation or compression code in Fast IPsec; this comes entirely from the FreeBSD cryptographic framework.

2.1. Protocols

The protocol implementations in Fast IPsec are completely different from those found in KAME. They borrow heavily from what is found in OpenBSD in several ways:

1) The protocols are structured in a "continuation style" that permits the decoupling of cryptographic and protocol processing. That is, each of the input and output processing paths for AH, ESP, and IPCOMP are broken up into "before" and "after" portions that invoke the cryptographic framework and then continue processing on return through a callback mechanism.
2) Many code paths are unified to handle both IPv4 and IPv6 protocols. This eliminates code duplication.
3) IP-in-IP encapsulation is implemented as a separate protocol. KAME handles the encapsulation/decapsulation of packets for tunnels as special case code in the ESP and AH protocols.
4) Header data are retrieved from packets with "m_copydata" instead of forcing mbuf data to be contiguous. This eliminates many assumptions about the handling of data in layers above and below. Performance measurements indicate doing this adds no noticeable cost over the techniques used by KAME.

Otherwise, notable differences in the protocols are in areas like statistics. Fast IPsec is careful to record error statistics that uniquely identify each problem; this is critical for diagnosing systems without source code.

2.2. Packet Tags

Aside from the cryptographic framework that is described elsewhere [Leffler, 2003], Fast IPsec required only the addition of "packet tags" to FreeBSD. Packet tags are used to associate typed variable-length data with a packet. The standard mbuf routines propagate these tags when packet headers are moved or duplicated and they are automatically reclaimed when an mbuf chain is freed. Packet tags originated in OpenBSD [Keromytis, 2003] but they were changed in two important ways:

1) OpenBSD allocates all storage with **M_NOWAIT** (do not block to wait for memory). This is too inflexible for general use. In FreeBSD routines that allocate tag storage take a parameter that specifies whether the caller is willing to block when allocating storage.
2) OpenBSD defines packet tag type values as a single 16-bit value that must be centrally administered to ensure uniqueness. In FreeBSD this 16-bit value was expanded to add a 32-bit **cookie** that specifies an ABI or module ID. By convention cookies are defined as the date and time that a module is created, expressed as the number of seconds since the epoch (e.g. using the output of "date -u +%s"). This scheme permits software modules to be developed without the need for a central administrator [Elischer, 2002]. Developers define a unique cookie and then manage tag types privately.

When these changes were made, OpenBSD-compatible shims were provided so that cross-platform compatibility could be maintained in code that needed to be portable. Packet tags were also used to replace the KAME "auxiliary mbuf" mechanism.

2.3. Security Database

The security database and policy management code is derived from the KAME implementation. While numerous changes were done to reduce memory usage and improve performance, its design has been left mostly intact.

The most notable change was to replace the use of **sockaddr_storage** for recording network addresses with a **sockaddr_union** data structure that is a union of the potential address formats. This reduces the storage for a network address from 128 bytes to 28. For systems with limited memory this is significant as each entry in the security database records at least two addresses. This change also reduces the size of the runtime stack and the amount of data referenced by structure copy and zeroing operations.

Otherwise the explicit algorithm specifications and algorithm-related data were replaced with references to transformation routines through which the protocols interact with the cryptographic subsystem. This technique has been used successfully in other systems [Keromytis et al, 1997; Spencer et al, 2002].

2.4. Locking

Fast IPsec was initially developed for the 4.6 release of FreeBSD. This version of the operating system uses traditional synchronization techniques designed for a non-preemptive uniprocessor environment. Only one thread of execution is expected to be active at a time. Code that needs to synchronize access to data structures does this by blocking interrupts so that asynchronous events are disabled.

FreeBSD 5.0 is the first release of FreeBSD to have a fully preemptive kernel. In this environment it is undesirable to guard execution paths to ensure synchronization of data structures [Hsu, 2003]. Instead locks are associated with data structures and concurrent threads of execution are managed by preempting a thread when it encounters a locked data structure. This can simplify code and make proper locking more intuitive.

Fast IPsec has very few locking requirements. The protocols are expressly designed to function as separate threads that depend only on private data. The only central data structure that needs synchronization is the security database. The database is referenced by code that is executed on behalf of system calls (applications sending data), by asynchronous activities that occur because of incoming network traffic, and by timer-driven threads that do things like expire security policies. Prior to FreeBSD 5.0 it was possible to block asynchronous access to the security database by raising the processor priority to **splnet**, but with a fully preemptive kernel this no longer works. Instead a set of locks were added to guard accesses to each of the major data structures and references to data structures are safeguarded with reference counters. These changes were straightforward and sufficiently fine-grained that there is no noticeable lock contention (as measured by the lock profiling facilities). The only real issue is the checking of security associations when transmitting packets. Due to the structure of the database inherited from KAME it is necessary to lock the **ipsecrequest** structure to ensure references to security associations are safely held. This turns out to be a "hot spot" that limits performance for bidirectional traffic flow. A redesign of the data structures to eliminate this locking is in progress.

3. Performance Analysis

There are several major areas to study to understand the performance of Fast IPsec: the cryptographic subsystem, the protocol implementations, the security database, and the network interface drivers. (There are other components such as mbuf and memory allocators but their individual performance tends to be less critical.) By far the majority of the overhead associated with IPsec is in the cryptographic processing. This is why it is so important to accelerate and offload the work from the main CPU. However the interactions between the various software components and the system architecture can have a noticeable effect on overall performance too. The next sections discuss these issues in more detail.

3.1. Crypto Subsystem

Fast IPsec uses the FreeBSD cryptographic framework to do all encryption and authentication work. This subsystem provides general-purpose device-independent support for a variety of transformations, including the symmetric-key cryptographic operations required by the ESP and AH protocols. The crypto support is comprised of a core set of code that manages requests and a set of device drivers for cryptographic devices. In addition there is a software-only device driver that implements symmetric-key operations on the host CPU for systems that do not have hardware devices.

The payload of each packet sent and received is passed to the crypto subsystem. Prior to dispatching the crypto request all the data needed to process the packet on return are collected and associated with the request. When the operation completes the crypto subsystem invokes a callback function stored in the request. This callback method completes the processing for the packet and dispatches the packet either "up" (for reception) or "down" (for transmission).

[Leffler, 2003] describes the work that was done to optimize the performance of the FreeBSD cryptographic subsystem. Increased performance of the crypto subsystem directly affects the performance of Fast IPsec. Of particular note is the work done to reduce latency in processing cryptographic requests. Compared to other systems, the FreeBSD cryptographic support has significantly less overhead and lower latency. This is reflected in significantly higher performance, especially for embedded systems where CPU cycles spent on overhead are more noticeable and for high-end systems where keeping the cryptographic hardware busy is critical to optimal performance.

3.2. Data Handling, Alignment, and Fragmentation

The performance of network protocols is typically constrained by the efficiency with which data are

moved (or not moved) and manipulated. Because IPsec requires significant computation to process each packet, inefficiencies in this area can be less noticeable. Nonetheless, while tuning the performance of Fast IPsec data handling issues frequently appeared.

One issue was the need to properly align packet data to ensure it is always processed with the "fast path." For example, some network interface drivers did not properly align received Ethernet frames so that the IP header is aligned to a 32-bit boundary. This can force data to be copied by protocols as the packet is passed up the stack. Further, when this data is passed to a crypto device driver, misaligned data can require additional copying. Several drivers with problems of this sort were fixed and the Fast IPsec protocols take care to ensure data are optimally aligned for ancillary operations. Fast IPsec and the FreeBSD cryptographic framework also keep statistics on any misaligned or otherwise suboptimal data manipulations.

Optimizing the input data path is simpler than optimizing the output path. For devices with a fixed-size link-level header, drivers can set up receive buffers so that data are contiguous and well aligned. As packets work their way up the protocol stack protocol headers can be efficiently removed. The only issue is ensuring proper alignment of data that requires cryptographic processing and this happens automatically if the IP header is aligned to a 32-bit boundary.

Optimizing the output path is a bit more involved. Headers must be prepended and packets must be rewritten with cryptographic transformations. Data that comes from a stream socket typically must be copied to create a writable version that can be transformed. Other data, such as packets being forwarded by an IPsec terminator, may be writable and not need to be copied. Fast IPsec creates a writable copy of an mbuf chain with the "m_clone" routine. This routine uses an *aggressive coalescing* technique that tries to compact the resulting mbuf chain and linearize the data, but balances this goal against the cost of copying already writable data. Compacting an mbuf chain is good in that it reduces the number of individual segments that must be processed; especially when arranging DMA to/from cryptographic hardware devices. Linearizing data is critical to the efficient use of cryptographic hardware that does not support scatter/gather DMA; but it can also improve performance of software algorithms. The scheme employed by "m_clone" uses the following rules:

1) "Inline mbufs" are coalesced only when there is a cluster immediately preceding that has space to hold the data.
2) Mbufs with writable external storage are left untouched.
3) Mbufs with read-only external storage must be copied. If there is space in the immediately preceding mbuf, then the data are copied. Otherwise, new storage is allocated and the data are copied. Data larger than a cluster is broken into multiple mbufs.

This scheme is designed for Ethernet traffic where the maximum frame size fits in a cluster. Further it depends on the ability to identify whether or not mbufs with external storage are writable. Finally, the "busting of jumbograms" is required because many drivers assume packets can be directly mapped for DMA.

In practice many packets are coalesced into a single mbuf with all the data linearized. Packets being forwarded are easily identified as writable in FreeBSD 5.0 but require some special handling in 4.x versions of the system.[1] Statistics kept on the operation of this scheme indicate about 45% of the packets that require copying are coalesced into a single mbuf; but these numbers are believed to be artificially low. (Most traffic is from performance benchmarks that run on the gateway machine. In this case traffic is TCP-based and the data are received locally instead of being forwarded. Both these factors affect the results.) The issue with mapping packets for DMA is discussed below.

3.3. Network Interface Drivers

The operation of the network interface driver can strongly influence performance. Fast IPsec was tested with a wide variety of hardware and drivers. Performance varies significantly depending on load and frame size. The currently preferred device is the Intel PRO/1000 which comes in 32-bit and 64-bit PCI configurations.

Two issues with the driver in the handling of large packets were identified and fixed. First, the driver did not support the "bus dma" API for mapping mbuf chains on to the system bus for DMA. This meant that packets larger than a physical page had to be broken up into multiple mbufs (as described above for the "m_clone" function). The driver was redone to use the bus dma functions so that outbound packets may be left intact (though they presently are still broken up into clusters).

[1] In FreeBSD 5.0 read-only mbuf chains are explicitly marked, but in 4.x one must check a reference count that may be maintained in a driver-private data area.

The second issue was that received jumboframes were written to multiple segments and an mbuf chain was then constructed. An alternative approach is to allocate contiguous receive buffers. This however can require pre-allocation of a significant amount of memory because the device constrains receive buffer sizes to be a power of two (so a 9000-byte mtu requires 16 kilobyte receive buffers). The driver was changed to optionally allocate contiguous memory for receiving large frames.

This revised driver performs noticeably better than the standard one. With a 9000-byte mtu set on both interfaces of systems connected by a cross-over cable, **netperf** performance results with the standard driver drop by more than 210 Mb/s while performance for the driver with contiguous receive buffers increases by a modest amount (the increase is small because performance is already near peak and one of the machines is constrained by PCI bus bandwidth.)

3.4. System I/O Performance

Performance is influenced by many system-level issues. Cryptographic requests directed to hardware devices require two trips across the I/O bus for each operation. This means that I/O performance is very important in understanding a system's capabilities. Bus width, latency, and device configuration can significantly affect performance results. Kernel profiling indicates that most systems are limited by their ability to field and process interrupts. In an IPsec terminator configuration each packet requires four DMA operations (two for NIC DMA and two for hardware crypto DMA) and as many as four interrupts to be processed. This implies that interrupt overhead must be minimized for the highest possible performance. In particular IRQ multiplexing must be avoided as well as system overhead like harvesting "IRQ entropy" for the pseudo random number generator (PRNG). Polling techniques such those available in FreeBSD [Rizzo, 2001] can be useful in reducing this overhead. Some vendors of hardware crypto products claim to intelligently coalesce interrupts in their hardware but it is unclear if these are anything more than marketing hyperbole.

With regard to the PRNG, most cryptographic hardware includes a hardware random number generator (RNG) that can supply sufficient entropy to seed the system PRNG. While this can eliminate expensive techniques used to collect entropy data, care must be taken to evaluate the quality of the entropy supplied by the hardware. **Rndtest** is a kernel module that was developed to test the quality of random data sources [Wright & Leffler, 2002]. It monitors data on the way to the system PRNG to see if it complies with the FIPS 140-2 standard [Federal Information Processing Standards, 2002]. If data fail any of the FIPS 140-2 tests, they are discarded and no data from the source are passed to the system PRNG until compliant data are seen. Testing can be done continuously or periodically and there are various controls (e.g. whether to report problems to the system console) and statistics maintained. This facility has shown, for example, that the entropy data produced by a Broadcom BCM5822 device is sometimes unreliable.

4. Performance Results

The IPsec protocols are very flexible. There are three protocols that can be combined to generate a variety of packet formats. Encapsulation may also vary: there is a **transport** mode in which the IPsec protocol headers are directly encapsulated in IP and a **tunnel** mode in which an additional IP encapsulation is done when an intermediate machine acts as an IPsec **terminator** or **gateway**. Tunnel mode is typical of IPsec-based Virtual Private Network (VPN) applications.

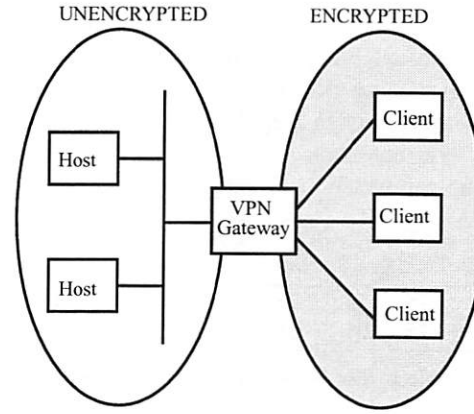

Figure 1: *Typical VPN gateway configuration*

For this paper we analyzed the performance of IPsec operating in tunnel mode using only the ESP protocol, but doing both encryption and authentication of the payload. When IPsec is deployed in this manner the network is typically configured as a VPN gateway, as shown in Figure 1. Specifically, IPsec data are received from peers, decrypted and authenticated, and forwarded in the clear to trusted clients. This means each packet requires one pass through cryptographic processing before it is forwarded. This is in contrast to a point-to-point tunnel configuration where data are cryptographically processed twice. Throughput measures for an IPsec gateway will be significantly higher than for a point-to-point tunnel. However collecting

performance data for a gateway configuration is more complicated because it requires multiple clients and/or peers to saturate a Fast IPsec gateway. With multiple clients and/or peers it is also necessary to include a switch in the configuration and this device can become a bottleneck when evaluating performance. Therefore, to ensure others can easily reproduce the results presented here, we have chosen a network configuration in which two machines are physically connected with a cross-over cable. Testing done on a multi-client terminator configuration confirms the result for this simpler configuration apply directly.

The test configuration has two systems, A and T, connected by a cross-over cable. Machine A has an Asus P4B533-V Intel845G motherboard with a 1.8 GHz P4 processor. A dedicated Intel Pro/1000 (82540) NIC located in a 32-bit PCI slot was used for testing. Machine T has a Tyan S2707G2N motherboard with a 1.8 GHz P4 processor. The Tyan system has dual Intel gigabit Ethernet devices (82545 and 82551) onboard. The 82545 has a 64-bit PCI interface and the 82551 has a 32-bit PCI interface. Testing was done using the 82545. The mtu on each interface was the default, 1500 bytes.

To establish baseline performance, tests were first run on an open network. Table 1 shows data collected using **netperf** version 2.2pl3. The data were collected with a 99% confidence interval (+/- 2.5%), identical buffering parameters, and a 32 kilobyte message size.

A->T			T->A			Operating
Mb/s	%Sys		Mb/s	%Sys		System
627	75	55	799	84	78	FBSD 4.8†
519	75	24	653	48	32	OBSD 3.3‡

Table 1: *Raw netperf results for fast testbed*

Next a manually-keyed IPsec tunnel was set up between the two machines. All traffic was encrypted with 3DES and authenticated with SHA1. Table 2 shows results for several configurations. The transfer rates reported by netperf are provided as well as the percentage of system time for the sending and receiving machines, as reported by the **vmstat** program. Performance is different depending on which machine is initiating the transfer; "A->T" results are for when netperf was run on the Asus-based machine, while "T->A" is when netperf was run on the Tyan-based machine. In one configuration a Broadcom BCM5822 was placed in each machine. Broadcom data sheets claim the 5822 can do 3DES+SHA1 calculations at 500 Mb/s. Testing described in [Leffler, 2003] measured a peak performance for a 5822 of 470 Mb/s when located in a 64-bit PCI slot and about 410 Mb/s when located in a 32-bit PCI slot. In the other configurations crypto calculations were done by the host CPU; this is denoted by "sw/".

† FBSD 4.8 is a July 4, 2003 snapshot of FreeBSD.

‡ OBSD 3.3 is a March 17, 2003 snapshot of OpenBSD.

A->T			T->A			Operating	Crypto
Mb/s	%Sys		Mb/s	%Sys		System	Support
160	80	73	170	70	94	FBSD 4.8	BCM5822
n/a	-	-	n/a	-	-	OBSD 3.3	BCM5822
43	100	93	44	89	97	FBSD 4.8	sw/FastIPsec
42	95	95	43	100	100	FBSD 4.8	sw/KAME
27	87	94	27	88	96	OBSD 3.3	sw/OpenBSD

Table 2: *IPsec tunnel results for fast testbed*

The results show Fast IPsec is about 60% faster than OpenBSD when using software crypto and slightly faster than the KAME implementation. The higher performance relative to OpenBSD shows the value of FreeBSD's optimized cryptographic subsystem. The comparison to KAME demonstrates that moving the crypto support out of IPsec into a general-purpose framework can be done without losing any performance.

Performance comparisons for Broadcom-accelerated configuration were not possible. KAME does not support hardware crypto acceleration and while OpenBSD claims support for the 5822 it did not work; traffic stalled and the tests never completed. However, based on the performance for raw cryptographic operations reported in [Leffler, 2003] and the relative network performance over the unencrypted link, one can assume Fast IPsec will be significantly faster than OpenBSD.

Performance in this simple network configuration is limited by the slowest component in the loop; in this case the CPU is the limiting factor. Tests with faster processors and the 5822 show performance of FreeBSD scales linearly with the CPU speed up to 218 Mb/s for 2.53 GHz P4 processors. At that point I/O bandwidth to the 5822 in Machine A becomes the limiting factor. 64-bit PCI support on Machine A would make a significant difference in the results.

Because no direct comparison to OpenBSD was possible with the Broadcom hardware a second testbed was set up with different hardware. In this configuration two slower systems were used, each with a GTGI XL-Crypt card. The XL-Crypt uses a Hifn 7811 crypto part and can do 3DES+SHA1 calculations at approximately 145 Mb/s. The two systems were purposely slower to match the performance of the cryptographic hardware (using slow crypto hardware in a

fast machine yields uninteresting results because the cryptographic hardware becomes the bottleneck.) Machine D was a Dell XPS 300 with 266 MHz PII processor. Machine E was an E-Machines etower 366i with a 366 MHz PIII processor. As before the systems were connected by a cross-over cable but this time only 100 Mb/s Ethernet was used. Table 3 shows the results for running netperf over a manually-keyed IPsec tunnel on this hardware.

D->E			E->D			Operating	Crypto
Mb/s	%Sys		Mb/s	%Sys		System	Support
45.9	81	20	51.4	92	22	FBSD 4.8	Hifn 7811
37.3	96	40	33.0	81	30	OBSD 3.3	Hifn 7811
9.8	100	93	9.4	99	92	FBSD 4.8	sw/FastIPsec
8.9	100	94	9.2	100	94	FBSD 4.8	sw/KAME
5.3	100	84	7.0	89	98	OBSD 3.3	sw/OpenBSD

Table 3: *IPsec performance results for slow testbed*

FreeBSD was 23/55% faster than OpenBSD when using hardware acceleration. With crypto operations done in software the difference was 34/84%. As the system performance decreases the FreeBSD optimizations are less noticeable because other factors become significant. In this case the I/O performance of the Dell system is so poor that it is the critical factor in determining overall performance.

5. Status and Future Work

An initial version of Fast IPsec was completed September 2001. Integration of this work into FreeBSD was completed November 2002 and committed to the stable branch in January 2003. The work described here, except for the fine-grained locking, is freely available as part of the FreeBSD 5.0 and 4.8 releases. Fast IPsec is currently being integrated into the NetBSD operating system [Stone, 2002]. Several vendors have incorporated Fast IPsec in their products.

Fast IPsec lacks support for IPv6; this has yet to be done because of lack of interest. The **IPCOMP** support is fully implemented but is not working due to an issue with the **gzip** compression support in the FreeBSD kernel.

The security database implementation needs to be redone. Much of the information stored in this database is better co-located with the routing tables so that the routing table lookup algorithms can be used. In addition much of the data stored in the database requires two and three levels of indirection to access. Relocating this data will simplify the protocols and eliminate some of the locking currently required to ensure indirect pointers do not change while the protocol code follows indirect references. A redesigned database is likely to provide a noticeable performance improvement for systems of any significant scale.

There is no support for the cryptographic algorithms required by the legacy versions of the AH and ESP protocols. Correcting this requires changes to the cryptographic framework API and may not be worthwhile.

The **PF_KEY** support used to communicate with user-level applications is intertwined with the IPsec security database implementation. This needs to be changed by separating the pfkeyv2 protocol and database implementations. This would fix, for example, problems where changing aspects of **PF_KEY** break user-mode applications such as **racoon** and **setkey**.

Otherwise, the most significant deficiency in the current design is the inability to use protocol-specific hardware operations. Most vendors of crypto hardware optimize their products for use as "all-in-one" devices that take an IPsec packet and parse the protocol and perform the cryptographic transforms in a single request. This is incompatible with the current general-purpose API provided by the cryptographic framework. Supporting this kind of operation requires exposing IPsec state that is currently private. However adding this is the only way that some hardware devices can be used at all as they do not otherwise provide access to the cryptographic transformation hardware.

6. Conclusions

Fast IPsec is an implementation of the IPsec protocols that has been designed as a plug-compatible successor to the KAME IPsec protocols with high performance. The software has been demonstrated to have performance more than 50% better than any other publicly available IPsec implementation.

7. Acknowledgements

This work incorporates ideas, and some code, from the OpenBSD IPsec implementation; without it this work would probably not have been done. Angelos Keromytis was helpful in understanding how bits and pieces of OpenBSD worked.

Vernier Networks supported this work; their contribution is greatly appreciated. Doug Ambrisko of Vernier Networks contributed many bug fixes and was a great help in testing.

References

Ambrisko, 2003.
D. Ambrisko, *Private email: 400mbs IPsec numbers* (February 2003).

Communications, 1999.
RedCreek Communications, Inc., *RedCreek IPSec VPN Card,* Source for the driver is included in Linux starting with 2.4 (April 1999).

Elischer, 2002.
J. Elischer, "Re: CFR: m_tag patch," *freebsd-arch@freebsd.org* (November 2002).

Federal Information Processing Standards, 2002.
Federal Information Processing Standards, "Security Requirements for Cryptographic Modules" (FIPS PUB 140-2), National Institute of Standards and Technology (December 3, 2002).

Hsu, 2003.
J. Hsu, "Reasoning about SMP in FreeBSD," *BSDCon 2003* (September 2003).

Kent, 1998.
S. Kent, "IP Authentication Header," *RFC 2402* (November 1998).

Kent & Atkinson, 1998a.
S. Kent & R. Atkinson, "Security Architecture for the Internet Protocol," *RFC 2401* (August 1998).

Kent & Atkinson, 1998b.
S. Kent & R. Atkinson, "IP Encapsulating Security Payload (ESP)," *RFC 2402* (November 1998).

Keromytis, 2003.
A. Keromytis, "Tagging Data In The Network Stack: mbuf_tags," *BSDCon 2003* (September 2003).

Keromytis et al, 1997.
A. Keromytis, J. Ioannidis, & J. Smith, "Implementing IPsec," *Proceedings of Global Internet (GlobeCom) '97,* pp. 1948-1952 (November 1997).

Leffler, 2003.
S. Leffler, "Cryptographic Devices Support for FreeBSD," *BSDCon 2003* (September 2003).

Rizzo, 2001.
L. Rizzo, "Polling versus Interrupts in network device drivers," *BSDConEurope 2001* (November 2001).

Spencer et al, 2002.
H. Spencer, R. Briggs, D. Redelmeier, M. Richardson, S. Harris, & C. Shmeing, *Linux FreeS/WAN* (April 2002).

Stone, 2002.
J. Stone, *Private email: Re: Fast IPsec: patches for FreeBSD 4.x?* (November 2002).

Wright & Leffler, 2002.
J. Wright & S. Leffler, *rndtest* (2002). http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/dev/rndtest.

Biography

Sam Leffler has been actively working with UNIX since 1975 when he first encountered it at Case Western Reserve University. While working for the Computer Systems Research Group (CSRG) at the University of California at Berkeley he helped with the 4.1BSD release and was responsible for the release of 4.2BSD. He has contributed to almost every aspect of BSD systems; most recently working (again) on the networking subsystem. You can contact him via email at <sam@errno.com>.

The WHBA Project: Experiences "deeply embedding" NetBSD

Jason R. Thorpe
Allen K. Briggs
Wasabi Systems, Inc.

Abstract

Traditionally, use of BSD in an embedded application has been limited to "stand-alone" environments such as server appliances, wireless access points, and the control plane of routing equipment. Recently, Wasabi Systems was given the opportunity to use NetBSD in a more "deeply embedded" application, as the software on a PCI storage adapter card. This paper will introduce our specific application and describe the requirements of the storage adapter environment, the challenges presented by this type of environment, and the solutions we have developed to address these challenges, along with the results of our effort and some outstanding issues that warrant further study.

1. Introduction

BSD has been used in embedded systems for several years in applications such as server appliances and network infrastructure equipment. These applications tend to take advantage of BSD's traditional strengths, namely networking and file systems. However, these applications present environments not all that different from the traditional stand-alone UNIX system.

Recently, Wasabi Systems, Inc. was given the opportunity to use NetBSD as the firmware on a PCI storage adapter card. This "deeply embedded" application is very different from traditional BSD embedded applications.

The goal of the WHBA (Wasabi Host Bus Adapter) project is to produce a design for a PCI add-in card that can be used to offload a variety of processing chores from the host system, which is typically a server. The initial application is that of an iSCSI target HBA. An iSCSI target HBA performs TCP/IP, IPsec, and iSCSI protocol processing, then passes the SCSI command and data payload up to the host system. Once the host system has processed the SCSI command, the HBA encapsulates the status code and any payload or sense data as necessary, and then sends the result back to the initiator. This is very similar to a parallel SCSI or Fibre Channel HBA operating in target mode.

Other potential applications for the WHBA include iSCSI initiator, IPsec offload processor, RDMA processor, TCP offload engine (TOE), and RAID controller.

2. WHBA requirements

The requirements of an HBA application are much different from those in a traditional embedded BSD environment. Here we will discuss the requirements of our application, as well as the general requirements and challenges of the HBA environment.

2.1. Overview of iSCSI

iSCSI is a transport layer for the Small Computer System Interface command set that uses TCP/IP to connect initiators and targets over standard IP networks. There are several usage models for iSCSI, including:

- Expandable direct-attached storage for consumer and SoHo environments
- At-distance connectivity to disk mirrors and tape volumes
- IP-based Storage Area Networks (iSANs)

Each of these usage models has different characteristics and performance requirements (described below).

The main advantage of iSCSI is its ability to run on any IP network, including LANs and WANs. Since IP-capable networks are ubiquitous, the cost of deploying iSCSI is low, especially compared to the cost of Fibre Channel; in addition to the cost savings from not having to deploy a different type of network (Fibre Channel), every server and client system on the market today comes with an Ethernet port. In environments where a SAN is not needed, iSCSI still has the advantage of not requiring the storage to be located within a few meters of the system to which it is attached.

iSCSI's basic unit for data transfer is a *protocol data unit*, or PDU [1]. There are several PDU types, but they all share the same basic structure:

- A *basic header segment* (BHS) that contains the PDU's opcode, the LUN, task tag, CDB, or other opcode-specific fields
- Zero or more *additional header segments* (AHS)
- An optional *header digest* (CRC32C)
- An optional *data segment*
- An optional *data digest* (CRC32C)

The maximum PDU size is negotiated when an iSCSI session is established. All data transfers are broken down into one or more PDUs. Since a common PDU size is 8KB, it is likely that any given SCSI data transfer will require multiple PDUs.

In addition to the overhead of encapsulating into or de-encapsulating data out of PDUs, the iSCSI transport has an optional error recovery mechanism, known as *markers* [2], that requires additional processing. Markers are inserted into the data stream at regular intervals, and allow an iSCSI target or initiator to re-synchronize the stream in the event a PDU with an invalid header digest is received (the length fields in the basic header segment can not be used in this case). These markers are orthogonal to PDUs, and thus are not accounted for in length fields or the header or data digests.

The combination of TCP/IP processing, PDU encapsulation and de-encapsulation, CRC32C generation and verification, and marker insertion and removal make a compelling argument for including an offload component in an iSCSI solution.

2.2. Usage model requirements

Each of iSCSI's usage models has its own set of characteristics and performance requirements. There are several variables to be factored in, including:

- Target audience and what they expect from a storage system.
- Acceptable cost of the storage system.
- Need for remote or local connectivity.

In order to understand the requirements of the HBA, it is helpful to have a basic understanding of some of iSCSI's usage models.

2.2.1. Consumer / SoHo

As homes around the world become "wired", the need for high-capacity storage increases. Consider the amount of space required to store music, pictures and movies, etc. Not only is the typical consumer PC limited in the amount of disk drives that can be directly attached, but the typical home PC user is not a computer hardware expert, and usually does not wish to open up his or her computer to install new disk drives.

A similar situation exists with the SoHo environment. Data stored on an office storage system is often business-critical, and tends to be stored for long periods of time. Therefore, as time goes on, more storage is needed. Because of the business-critical nature of this data, the downtime and risk associated with migrating to new, higher-capacity disks is unacceptable.

From the perspective of storage, consumer and SoHo environments are not generally high-performance environments. Low-end PCs and laptops, which are generally tuned for reduced power consumption, are common. The expectation of the user is that their files and pictures will be available on-demand, and their music and movies will play smoothly. The performance achievable with 100Mb/s Ethernet is adequate for audio applications. Gigabit Ethernet is becoming increasingly common on consumer PCs, and the iSCSI performance that can be achieved on Gigabit will likely be comparable to, if not better than, what the user would experience with disk drives directly attached to the computer.

In this usage model, cost is a dominant factor. In order for an offload solution to work in this market, a small number of low-cost parts should be used on the HBA.

2.2.2. At-distance backup

At-distance backup is a scenario where storage resides in one physical location and the capacity to back up that storage resides in another, possibly far away, location. There are a number of reasons why this strategy might be used, including cost-effectiveness (backup systems can be expensive) and disaster recovery (losing your data and its backup in the same fire would render the backup useless).

This environment is constrained by the characteristics of wide-area networks. The user will not expect low latency or high throughput access to storage, but will expect the data to stream effectively, and for software on the initiator (client) side to behave as though the storage were local, to the extent possible [3].

This usage model is less concerned with cost than the low-end consumer/SoHo market. However, small businesses are likely to be a major part of this market, and so keeping the cost down is still important.

2.2.3. Storage Area Networks

Storage Area Networks, or SANs, are typically deployed in environments where multiple systems require access to pooled storage, or where redundant connections to storage are required in order to ensure uptime.

SANs are currently dominated by Fibre Channel. However, Fibre Channel has a high total cost of ownership [4]:

- Fibre Channel is not commonly used for IP networks, and thus is always added cost.
- Fibre Channel equipment is expensive.
- IT staff generally needs additional training in order to manage and maintain a Fibre Channel SAN.

Because iSCSI runs over any IP network, it can utilize existing networking infrastructure and expertise, thus reducing the total cost of ownership.

Even though Fibre Channel is currently faster than Gigabit Ethernet (Fibre Channel is available in 1Gb/s and 2Gb/s speeds), the reduced cost of iSCSI still makes a compelling argument in favor of its use. However, users of Fibre Channel have come to expect low latency and high throughput from their SANs, largely due to the fact that Fibre Channel is a fairly low-overhead protocol and Fibre Channel HBAs offload all of the necessary processing.

Even an expensive iSCSI HBA is likely to be less expensive than a Fibre Channel HBA; current iSCSI HBAs retail for approximately $500US, and Fibre Channel HBAs retail for $800US and up. Combined with the cost savings associated with the use of commodity Gigabit Ethernet networking equipment, iSCSI presents a much lower-cost alternative for SAN deployment. As such, the cost of an iSCSI HBA is less of a factor in this usage model.

2.3. Host communication

An HBA plugs into a host system and must communicate with it. HBA applications based on intelligent I/O processors generally use a *messaging unit* and a DMA controller to implement host communication.

A messaging unit is a device in an I/O processor that provides support for notifying the host and the HBA when new messages are available. A typical messaging unit is comprised of two or more *doorbell* registers and two or more sets of *queue pointers*.

The queue pointers in the messaging unit are used to implement a ring of message descriptors. These message descriptors reside in host memory. The DMA controller is used to copy the host-resident descriptors to HBA local memory. The DMA controller is also used to move payload data between the HBA and host.

We looked at several examples of message passing APIs used by storage controllers. Most were either too simplistic (early BusLogic and Adaptec), too complicated to implement in a short period of time (I2O), or too tied to their specific hardware platform and application (recent Adaptec). Since one of our goals was to produce a solution that could offload different types of workloads, we decided to borrow ideas from the early BusLogic [5] and the I2O [6] APIs and create our own.

The WHBA message passing API is based on two circular queues; one for host->HBA messages and one for HBA->host messages. The general data flow is that the host sends a message to the HBA and at some time in the future, the HBA responds to that message, although there are a few exceptions to this rule. Messages and replies use the same data format.

Each message is 64 bytes long and begins with a standard header:

```
uint32_t msgid_high;
uint32_t msgid_low;
uint16_t msg_type;
uint16_t hdr_reserved;
```

The `msgid_high` and `msgid_low` fields are not processed by the WHBA, but are copied into the corresponding fields of message replies, where they may be used by the host's driver software to identify the context associated with original message sent by the host to the HBA. The `msg_type` field defines the remaining data fields after the header. The `hdr_reserved` is reserved for future expansion.

There are currently four groups of messages defined in the WHBA message passing API:

- General messages
- iSCSI common messages
- iSCSI initiator messages
- iSCSI target messages

Some of these messages are self-contained, and others contain pointers to additional message data. This data is then transferred using the DMA controller from the host's memory to the HBA's memory. Either the message or the additional message data may also contain pointers to payload data associated with the message.

2.4. HBA hardware

Two hardware platforms, both based on Intel's XScale™ processor core, were used during the development of the WHBA:

- The Intel IQ80321 reference platform for the i80321 I/O Processor
- The ADI Engineering *i*/HBA (which was designed and built expressly for the purpose of demonstrating the WHBA)

The IQ80321 was used for early prototyping and development. This board consists of the i80321 I/O Processor, an Intel i82544 Gigabit Ethernet controller, an IBM PCIX-PCIX bridge, a DIMM slot, 8MB of flash memory, and a UART. The i80321 contains the XScale™ core, PCI controller, DMA controller, timers, and an I2O-compatible messaging unit.

The ADI *i*/HBA was the target platform for the WHBA. The *i*/HBA consists of an i80200 CPU, an Intel i82545 single-port or an i82546 dual-port Gigabit Ethernet controller, an Intel i21555 non-transparent

PCI-PCI bridge, 128MB or 512MB of on-board SDRAM, 8MB of flash, and a Xilinx Virtex-II FPGA containing the companion chip, known as HBACC.

The HBACC is a descendant of the BECC ("Big Endian Companion Chip") used on ADI's BRH XScale™ development platform. The HBACC contains a high-performance memory controller, the PCI controller, a DMA controller with CRC32C offload capability, timers, and a UART. The i21555 non-transparent PCI-PCI bridge contains an I2O-compatible messaging unit.

Initial tests on each of these platforms showed raw TCP throughput of 300-400Mb/s, with the IQ80321 being on the low end of that range and the *i*/HBA being on the high end. Since a high-performance host could easily saturate either of these platforms, it is necessary to utilize the hardware available on the WHBA as effectively as possible.

The XScale™ processor is an implementation of the ARM5TE architecture. This processor has an MMU and virtually-indexed/virtually-tagged instruction and data caches. While the MMU provides some support for address space identifiers, the semantics of these identifiers and the requirements of the application preclude their use. As a result, process context switches carry a significant penalty: changing address spaces requires a complete cache clean and TLB invalidation.

In addition to context switch overhead, the memory buffer used to stage data to and from the host must be managed. The means of moving data in and out of this buffer is through a DMA controller, so extracting the physical address of any given region of the buffer is a time-critical operation.

Finally, the latency of application notification when a messaging unit doorbell or queue pointer register is written is critical.

2.5. System start-up and shutdown

System start-up and shutdown in the HBA environment are very different from start-up and shutdown in the desktop and server environments where BSD is traditionally found. These differences present some interesting challenges.

Most BSD systems rely on the system's firmware to bootstrap the hardware and read in a loader program from a storage device, such as disk or CompactFlash. In the HBA environment, there are no disks. While our prototyping systems had the RedBoot [7] bootstrap and debug environment in ROM, it is unlikely that any final product sold to end-users will include it. This puts the onus of hardware bootstrap entirely on the operating system, which must be loaded into ROM at the reset vector.

Furthermore, BSD systems have historically not been very fond of unexpected shutdowns, which are often associated with long restart times (due to file system checks) and lost data (due to data not being flushed out to disk, as would happen in a clean shutdown). In contrast, unexpected shutdowns are a part of normal operation in an HBA environment. These can be caused by a number of events on the host system including reboot, driver load/unload, and power management activity.

3. The Wasabi Embedded Programming Environment™

In order to meet the challenges before us, we concluded a traditional BSD operating model would not be appropriate for the following reasons:

- A good portion of our application needs to reside in the kernel in order to be able to interact with messaging and DMA hardware effectively.
- A large chunk of physically contiguous SDRAM is required for efficiency.
- We needed to address the start-up and shutdown issues associated with the HBA environment.

We observed that by placing the entire application into the kernel, user space became completely superfluous. Eliminating support for user space would free up a large chunk of space in the virtual address map that we could be used to contiguously map SDRAM. With no user-land, there is little need for a root file system, and eliminating the root file system goes a long way towards addressing start-up and shutdown issues.

Unfortunately, our existing iSCSI software was a pthreads application that ran in user space. What we needed was an application environment that would allow an application to run either in user space or the kernel. So, we developed one, and called it the *Wasabi Embedded Programming Environment*™, or WEPE for short.

WEPE is actually a combination of three things:

- An API for applications that provides portability to both user space and kernel WEPE environments
- A configuration management framework that eliminates the need for configuration files
- A set of modifications to the NetBSD kernel that removes support for processes running in user space and provides a simple, extensible debugging console

While there are many applications that can benefit from the simplified single address space model that WEPE provides in the kernel environment, it is important to recognize that there are significant trade-offs associated with using this model.

Perhaps the most significant of these is the lack of memory protection; a badly behaving application can corrupt data structures in another application.

Secondly, since all applications are linked together into a single ELF image, applications, as well as the kernel itself, must be careful with regard to symbol namespace.

Finally, the WEPE API must contend with the fact that the semantics of calling system facilities is quite different in the user space and kernel environments, and global state, such as `errno` cannot be used in the kernel environment. While thread-specific data can be used to work around many of the awkward POSIX API issues, such schemes can have problems with efficiency that vary from platform to platform. For this reason, WEPE does not provide a POSIX API, requiring that applications be ported to the WEPE API.

3.1. The WEPE API

In addition to the trade-offs listed above, there are some functions available in the user space environment that do not map directly to equivalent functions in the kernel. It is therefore necessary to provide applications with a compile-time indication of whether they will be run in user space or in the kernel environment. This indication is provided by the presence of a C preprocessor macro:

- `_WEPE_MODEL_SINGLE` indicating the single address space kernel environment
- `_WEPE_MODEL_VIRTUAL` indicating a full user space virtual memory environment

WEPE API elements are logically grouped by function, and have naming rules that help to identify their function and grouping. For example, functions whose names begin with `wepe_sys_` map to system calls. The following sub-sections provide an overview of these functional groups.

3.1.1. File I/O functions

WEPE file I/O functions are made available by including the `<wepe_fileio.h>` header file. This functional group provides WEPE versions of several Unix file-related system calls, such as *open(2)*, *read(2)*, and *close(2)*. All of these functions return 0 on success or an error code to indicate the reason for failure. Functions that would return a file descriptor or a byte count in a POSIX environment take an additional pointer to that returned value in WEPE.

3.1.2. Socket I/O functions

WEPE socket I/O functions are made available by including the `<wepe_sockio.h>` header file. This functional group provides WEPE versions of several Unix socket-related system calls, such as *socket(2)*, *bind(2)*, and *listen(2)*. Like the file I/O group, all of these functions return 0 on success or an error code to indicate the reason for failure, and functions that would return a file descriptor in a POSIX environment take an additional pointer to that returned value in WEPE.

As in POSIX, any file descriptor returned by a WEPE socket I/O routine may be used with file I/O functions.

3.1.3. Thread functions

WEPE thread functions are made available by including the `<wepe_thread.h>` header file. This functional group includes a threads API similar to, but simpler than, POSIX threads, including thread creation, mutexes, read/write locks, and condition variables. The thread group also includes some basic process management functions.

3.1.4. Networking functions

WEPE networking functions are made available by including the `<wepe_net.h>` header file. This functional group provides WEPE versions of several networking-related functions, such as *getifaddrs(3)*, *getaddrinfo(3)*, and *getnameinfo(3)*. In the kernel environment, this group also provides function calls that allow applications to configure network interfaces, set routing table entries, and access the kernel's DHCP client.

3.2. WEPE configuration management

While the configuration management framework used by WEPE is actually independent from WEPE, it bears mentioning in this context since WEPE relies on its functionality. The main problems it attempts to solve are as follows:

- In embedded systems, management and storage of multiple configuration files can be problematic, especially if there is no file system in which to store multiple files.
- Configuration files have differing syntaxes, and thus make it difficult to provide a consistent configuration interface.

The MIB model used by the BSD *sysctl(8)* tool is very attractive from a consistency point of view.

However, it does not define the storage mechanism for the configuration data.

The solution that WEPE employs, both in the kernel and in user space environments, is known as *wctdb* ("Wasabi Config Tool Database"). The database consists of simple key-value pairs. The keys are hierarchical MIB names, similar to those found in the *sysctl(8)* interface. The API is made available by including the `<wctdb.h>` header file.

The implementation of the back-end database is completely hidden from the user of the API. In user space, *btree(3)* is used. In the kernel environment, a simple key-value list designed for storage in memory-mapped flash is used.

3.3. WEPE kernel modifications

The WEPE kernel modifications fall into four categories:

- Replacement of several functions in the kernel proper in order to remove the user address space
- Implementation of the WEPE API
- Implementation of the WEPE debug console **kshell**
- Kernel environment debugging tools

In this section we will discuss the replacement functions, the **kshell** debug console, and the kernel environment debugging tools.

3.3.1. Replacement functions

One of our goals was to make WEPE as non-invasive to the rest of the kernel as possible. However, there are several places in the kernel proper that access the user address space, using routines such as `copyin()` and `copyout()`. By replacing these functions, we are able to use all of the system call functions in the kernel to provide WEPE API support without modifying them.

There is a slight problem with this approach, however. The semantics of these user address space access functions dictate that the data be copied. This is not strictly necessary in the kernel environment, since all tasks share a single address space. Unfortunately, many parts of the kernel assume that the arguments provided in a system call are "owned" by the code that implements the call, and thus may modify the arguments while executing the system call. For this reason, we must accept a certain amount of this data copying.

There is one other function that is replaced, although for a different purpose. The `start_init()` routine, which normally takes care of starting *init(8)*, is replaced with the entry point for the **kshell** debug console.

3.3.2. The kshell debug console

The **kshell** debug console provides a command-line interface to the WEPE environment. Its primary purpose is for debugging, but it is extensible and could be used to provide a command-line based management interface in an appliance application. The **kshell** is also responsible for initializing all of the applications that are linked into the kernel image.

The **kshell** provides several API components available only in the kernel environment. These API components are made available by including the `<wepe_kshell.h>` header file.

Kshell takes the place of *init(8)* as process #1. When it starts, its first task is to initialize the console device in a way that mimics the behavior of a normal tty; the initial (and only) session is created, the console device is set as its controlling terminal, and backspace is set to *^H*.

Once the console is initialized, **kshell** calls the initialization function provided by each application linked into the image. Applications register themselves using a C preprocessor macro: `KSHELL_APPINIT()`. This macro causes a pointer to the application's initialization function to be placed into a special section in the final ELF image. The **kshell** then traverses this special section, calling through each function pointer. This scheme not only allows applications to be linked into the image easily, but also facilitates binary-only distributions to licensees. Applications may perform any start-up tasks they require using this mechanism, including creating additional processes and/or threads.

Once all of the application initialization tasks have been performed, the **kshell** enters a command loop. The following built-in commands are provided: *about*, *help*, *ddb*, *reboot*, *halt*, and *poweroff*, the latter three being the equivalent of the user space commands of the same names.

The **kshell** also allows applications linked into the image to add commands to the command line interface. Applications register these commands using the `KSHELL_COMMAND_DECL()` C preprocessor macro. This macro causes a pointer to a structure describing the command to be placed into a special section in the final ELF image. This section is consulted by the **kshell** command loop when a command is entered on the command line interface. The command description includes the command name, a usage string, and a pointer to the function that implements the command.

The **kshell** exports some support functions that help application programmers to implement command extensions. These support functions include a *getopt(3)*-like routine and a paged-output routine.

3.3.3. Kernel environment debugging tools

Debugging tools for Unix systems are generally used in a user space environment; if your program crashes, you run *gdb(1)* on it to figure out why. Similarly, if you want to profile your application, you compile it for profiling, run it, and use *gprof(1)* to analyze the profiling data.

NetBSD also provides support for using debugging and profiling tools on the kernel: in-kernel (DDB) and remote (KGDB) debuggers are available, and the standard BSD kernel profiling feature *kgmon(8)* is supported.

Unfortunately, the standard BSD debugging and profiling tools were insufficient for the WEPE kernel environment for the following reasons:

- DDB is not a source-level debugger, and thus has limited usefulness for debugging a complex application.
- KGDB requires a dedicated serial port; the WHBA target hardware only has one, which is used for the console.
- Kgmon requires a user space application to control kernel profiling and to extract the profile data.

The debugging problem was addressed by enhancing the in-kernel DDB debugger to interact better with KGDB. When an event that traps into DDB occurs (either a special sequence on the console, the *ddb* command in **kshell**, or a fatal trap or kernel panic), DDB is entered as normal. If the user wishes to use KGDB to debug the problem, the new *kgdb* command is entered at the DDB command prompt. This causes the console port to become the KGDB port, disabling normal console output. The user may now interact with the application using KGDB. When the user is finished using KGDB, he or she simply disconnects the debugger and the port returns to console mode, presenting the DDB command prompt once again. At this point, any DDB command, including *continue*, may be issued.

The profiling problem required a more complex solution. We started by implementing a *kgmon* application for the kernel environment. This optional application is linked into the **kshell** command line interface using the standard **kshell** extension facilities, and includes the same functionality as the user space *kgmon(8)* utility. This allowed us to perform all of the necessary control operations for kernel profiling.

Once we were able to control kernel profiling, we needed a way to examine the profile data. Since our *gprof(1)* tool is cross-capable, we decided to add a TFTP client to the kernel in order to dump the profiling data over the network.

4. Implementation of the HBA application

The iSCSI application for the HBA breaks down into several components: startup code, the PCI interface, the iSCSI engine, and some glue that stitches the pieces together.

4.1. HBA startup

The start-up component of the HBA application varies from platform to platform. For example, the IQ80321 platform requires that bootstrap software configure the memory controller so that SDRAM is available, whereas the *i*/HBA's HBACC performs this task.

The primary challenge of HBA start-up is that the application requires certain PCI resources that must be configured by the host system. Meanwhile, the host must not configure the PCI resources until the HBA has completed its initial setup, such as setting of the size of the PCI base address registers that the host must program. This issue is handled differently on our two hardware platforms.

The IQ80321 uses a semi-transparent PCI-X bridge and the host system programs the i80321's PCI-X configuration space directly. The bridge may be set into a mode such that configuration cycles issued from the primary (host) side of the bridge will be locked out and retried by the host until software on the HBA releases the bridge. However, if the bridge locks out configuration cycles from the host system for too long, the host system may fail its boot-time configuration and power-on self-test (POST). Since the i80321's memory controller must be programmed and an ECC scrub performed, PCI BAR sizes must be programmed by reset vector code. Fortunately, the RedBoot bootstrap environment for this board configures the i80321's PCI BARs in a way that is compatible with our HBA application, which allowed us to skip this task in our start-up code.

The *i*/HBA, on the other hand, uses the i21555 non-transparent PCI bridge. This bridge is similarly configured to lock out PCI configuration cycles, allowing our application code to configure the bridge. Fortunately, the *i*/HBA does not require an ECC scrub, and thus can start the HBA software rather quickly.

The HBA software is stored in flash memory on both of the platforms we used. In order to minimize the flash footprint of the HBA software, a special loader called *gzboot* was developed to allow the software to be stored in compressed form. *Gzboot* can be prepended onto a flat binary image of a NetBSD kernel that has been compressed with *gzip(1)* and the resulting image can be written to flash memory. When started, *gzboot*

will then decompress the kernel into a pre-determined location in SDRAM and jump to it. Code to bootstrap SDRAM and configure PCI BARs can be placed into *gzboot*'s start-up code.

During development, the RedBoot bootstrap environment was used on both boards to provide initial start-up support. On the IQ80321, a RedBoot script is used to perform an unattended start-up of the HBA software. On the *i*/HBA, the RedBoot reset vector code will automatically jump to the HBA software written into flash, but to aid development, it will drop into the RedBoot command prompt if a test point on the board is shorted to ground at hardware reset.

4.2. HBA PCI interface

As mentioned, the PCI configuration is somewhat different between the *i*/HBA and the IQ80321. For development, though, we wished to minimize the differences visible to the application layer. In this section, we'll go into a bit more detail on how we work with the PCI interface on each card and how we create an abstraction to keep the application code itself away from the differences.

4.2.1. IQ80321 PCI interface

The host system sees the IQ80321 as two devices: the IBM PCI-X bridge and the i80321 I/O processor (on the secondary side of the bridge). Depending on the configuration of the board, the host may also see other PCI devices on the secondary side of the PCI-X bridge.

The IQ80321 has four inbound PCI BARs. The first BAR is set to its minimum size, 4KB, and maps the i80321's Messaging Unit (MU) into the host's PCI space. The next two BARs are disabled. The last BAR maps the IQ80321's local memory onto the PCI bus, providing a window for PCI devices under the i80321's control to access the entire range of the IQ80321's SDRAM.

Since the PCI-X bridge on the IQ80321 is semi-transparent, it is possible for the host system to access the PCI devices under the i80321's control. The IQ80321 has a set of switches that allow the on-board PCI devices to be "hidden" (by making its IDSEL line unavailable) from the host.

The DMA controller on the i80321 can address the PCI bus directly; no address translation is required. This means that the HBA software can directly use the PCI addresses provided by the host system. Unfortunately, the DMA controller does not support true scatter-gather; there is a single chain of buffer descriptors, each descriptor containing a source address, destination address, and length. Since the length applies to both the source and destination in that descriptor, the source and destination DMA segments must map 1:1. It is not very likely that this would ever happen with host-provided scatter-gather lists during normal operation, so it is necessary to buffer data to/from the PCI bus in a physically contiguous memory region.

4.2.2. *i*/HBA PCI interface

The host sees the *i*/HBA as a single device: the i21555 PCI bridge. On the secondary side of the bridge is a completely independent PCI bus with its own address space. There are mapping windows on both the primary side and secondary side of the bridge that allow devices on either side to access devices on the other. Transactions that pass through the bridge go through an address translation process.

While the DMA controller on the HBACC can address the PCI bus directly (like the i80321), the HBACC's DMA controller does not operate on the same address space as the host (unlike the i80321). This requires the HBA software to translate PCI addresses provided by the host.

Further complicating PCI access on the *i*/HBA is the fact that the i21555's upstream (secondary to primary) memory window is not large enough to map all of the memory that might be installed on the host system. The i21555 provides a look-up table that sub-divides the upstream memory window into *pages* and remaps those individual pages onto the host's PCI address space. Note that the HBACC DMA controller and the bridge's look-up tables are distinct, which requires us to put some knowledge of the look-up tables into the HBA application in the *i*/HBA case.

One fixed BAR on the i21555 is part of the first downstream memory window. The first 4KB of that BAR maps the local registers on the i21555, and it is through this BAR that the host accesses the doorbell registers to signal the *i*/HBA. Another 4KB BAR is configured for access to a page of HBA local RAM that is used for extra register space, as there are only a few general purpose "scratch" registers on the i21555. The upstream BARs and look-up tables are configured on the fly.

Wasabi was consulted numerous times during the design of the HBACC, and as a result, we had a fair amount of input on the design of the HBACC DMA controller. The HBACC DMA controller fully supports independent scatter-gather lists for both the source and destination, and thus does not require a physically contiguous staging area. The DMA controller also supports the iSCSI CRC32C algorithm, and can compute a full or partial CRC32C when copying data or in a read-only mode (where no destination is used).

4.2.3. Messaging Unit API

While the messaging units on our two hardware platforms are both I2O-compatible, the interface is somewhat different both from the host's and HBA software's perspective. In order to minimize the impact of these hardware differences on the HBA software, we developed a general framework for interfacing with messaging units called **muapi**.

Muapi is a session-oriented mechanism for sending and receiving messages through different types of messaging portals found in a messaging unit. The basic architecture of **muapi** consists of *back-ends*, *portals*, *sessions*, and *messages*. Back-ends present portals to the **muapi** framework. Each portal has an associated data type:

- *Doorbell* portals can indicate a small number of specific events, such as "new message available" or "error occurred", represented by a bit mask.
- *Mailbox* portals can pass single integer messages. Mailboxes can be used to implement handshake protocols for initialization, among other things
- *Message queue* portals consist of *head* and *tail* pointers. The message producer advances the head pointer, and the message consumer advances the tail pointer. The pointers are indices into a ring of message buffers. These message buffers are usually accessed using the DMA controller.

Clients of the API create a session associated with a specific portal. Clients send messages by calling `muapi_put_message()`. Clients may receive messages either asynchronously (using a callback) or synchronously (using `muapi_get_message()` or `muapi_sync()`, which waits for the callback to be invoked). Once a client has processed the message, it acknowledges the messaging unit by calling the `muapi_release_message()` function, which may clear a mailbox or advance a tail pointer.

4.2.4. Data mover API

As with the messaging unit, our two hardware platforms used different DMA controllers with different programming interfaces. We also wanted a single framework capable of moving data between SDRAM and the PCI bus, moving data from one region of SDRAM to another, and handling CRC32C offload.

The solution used in the HBA application is called **dmover**. **Dmover** provides a session-oriented mechanism for queuing data movement operations. The API is fully asynchronous, and clients are notified by a callback when an operation completes.

The basic architecture of **dmover** consists of *back-ends*, *algorithms*, *sessions*, and *requests*. Back-ends present algorithms to the *dmover* framework. Clients of the API create a session to perform a specific algorithm, at which times a backend is chosen and assigned to the session. The client then allocates requests, fills in the necessary information, and queues them with the session.

In order to provide maximum future expandability of the interface, algorithms are named using strings. Back-ends and clients that wish to interoperate must agree on a name for a given algorithm. For example, the main **dmover** algorithms used by the HBA application are *pci-copyin* (copy from PCI to SDRAM) and *pci-copyout* (copy from SDRAM to PCI).

Since the address spaces used by the CPU running the HBA application and the PCI bus are separate, **dmover** must also handle multiple data representations. The **dmover** API currently supports linear buffers (mapped into the kernel virtual address space), UIOs, CPU physical addresses, and 32-bit and 64-bit scatter-gather lists. The following usages are common in the HBA application:

- *pci-copyin* with a source buffer type of *sglist32* and a destination buffer type of *paddr*.
- *pci-copyout* with a source buffer type of *paddr* and a destination buffer type of *sglist32*.

4.3. HBA iSCSI engine

The iSCSI engine is largely untouched from the user space application, but we did rework a couple of routines to adapt them to the kernel environment.

A number of the functions used in the iSCSI application were already wrapped to allow the kernel and user space implementations to be different. For example, all uses of `malloc()` and `free()` in the iSCSI code were already using `iscsi_malloc()` and `iscsi_free()`. Calls to these wrapper functions were replaced with calls to the corresponding WEPE API functions.

The most obvious place the iSCSI engine required retooling, however, was in how it handled network I/O. The iSCSI engine code uses `struct uio` and the `readv()`/`writev()` family of functions to read and write vectors of data. While this works well in user space, and was possible to do in the kernel environment using WEPE, it is much more natural and efficient in the NetBSD kernel to use *mbuf* chains in those places. This was achieved by adding some data type abstraction in the iSCSI application code.

Since our iSCSI software had heretofore been targeted at stand-alone appliances, it included a full SCSI command processing engine and assumed direct access to the backing-store. Since SCSI command processing is performed by the host in the HBA case,

our iSCSI engine was changed so that a distinct boundary exists between the transport and command processing components, in order to allow either piece to be replaced. This has benefits beyond enabling the HBA application: it enables our appliance software to support other SCSI transports, such as Fibre Channel.

4.4. HBA glue

In addition to code that parses messages from the host system, the HBA application requires a certain amount of "glue" to interface to the iSCSI application. This glue layer forms the SCSI command processing half of the iSCSI engine.

The interface is designed so that the HBA handles as much iSCSI-specific processing as possible. The interface between the host and the HBA is therefore largely a SCSI interface with a couple of hooks to notify the host of new iSCSI connections and sessions.

After each new session is established or a session disconnects, the HBA notifies the host with a unique session identifier. Apart from those messages, the bulk of the interface is simply data movement.

When a new command is received, the HBA sends a message to the host with the SCSI CDB. The host responds with one of several commands:

- *SendDataIn*, indicating that the target is responding to a SCSI "data-in" command or phase. In this case, the message contains a scatter-gather list pointing to the data in host memory that should be sent to the iSCSI initiator.
- *ReceiveDataOut*, indicating that the target is ready for the data in a SCSI "data-out" command or phase. In this case, the message contains a scatter-gather list pointing to the space in host memory where the data from the iSCSI initiator should be placed.
- *ScsiCommandComplete*, which, when sent in reply to a new CDB, usually means an error condition was encountered. This message contains the status and response codes as well as any sense data that should be sent to the iSCSI initiator.

The principal job of the HBA glue code is to marshal these requests from the host with the data requested and provided on the iSCSI session. It also handles moving the data to and from the host, using the scatter-gather lists provided by the host and the **dmover** facility.

5. Results

We had several goals in embarking on this project. Our primary goal was to build a workable system for tackling these "deeply embedded" types of applications with NetBSD running on an XScale™-based platform. Secondary goals included efficient use of the hardware, maximum performance, and adequate tools to measure different aspects of system performance.

While it is difficult to fully quantify, we believe that we have met our primary goal. We have functional iSCSI target HBA prototypes built around our modified NetBSD and a public proof-of-concept demonstration was given at Storage Networking World in April 2003 using an IQ80321-based iSCSI target HBA prototype, in cooperation with Intel and DataCore Software. Unfortunately, we have been less successful in meeting the secondary goals.

After we established the basic functionality of the prototype system, we set out to look at performance. For testing the performance of the iSCSI application, we required two systems: the initiator and the target. The target system included a prototype HBA, driven by a pre-release version of DataCore's SANsymphony™ software with support for iSCSI. This software was running on a dual 2.4GHz Xeon system with 512MB of RAM running Microsoft Windows Advanced Server 2000. The iSCSI initiator system was a Dell Dimension 4500 running with the Intel Pro/1000 IP Storage adapter under Windows XP Professional. On the initiator system, we ran an I/O test application called *Iometer* [8]. The default Iometer workload settings were used.

With the settings we used, iSCSI traffic to and from the target peaked at approximately 7.5MB/s using both 100Mb/s and Gigabit Ethernet networks. As the speed was effectively constant at different wire speeds, we had to conclude that the limiting factor was not in the network path but either in the test structure or in the HBA application itself.

The entire HBA application is relatively complex. To get a clearer picture of the system behavior, we began looking at just the raw TCP/IP performance. For this test, we used the *i*/HBA mounted in the PCI slot on an ADI Engineering BRH board and the Xeon system above running NetBSD with a uniprocessor kernel. The test application used here was *kttcp*, a TCP performance testing tool similar to *ttcp*, except the kernel provides the data source and sink, thus enabling us to test performance in an environment similar to the HBA. On the Xeon system, the *kttcp* version in NetBSD's pkgsrc system in ***pkgsrc/benchmarks/kttcp*** was used; on the *i*/HBA, the same was ported to the **kshell** environment. While the *i*/HBA's low-level diagnostics are able to send raw Ethernet frames at near-line-rate speeds in external loop-back tests, we were only able to achieve a bit above one-third of that speed in basic intersystem testing (as noted in section 2.4). Obviously, we would expect some overhead for TCP/IP processing, but we

did not expect to see such a substantial difference in performance.

After profiling the *i*/HBA side of the *kttcp* test both with time-based profiling and with profiling based on the hardware-resident performance monitoring counters, there is no one clear cause for the poor performance.

Clearly, for this system to progress from "workable" to "commercially viable" on this class of hardware, the performance issues will have to be addressed. However, considering the short amount of time we had in which to produce a working demo, we feel that the effort was a success.

6. Conclusions

In this paper we have described an embedded application outside the realm of traditional BSD embedded applications. We have shown the requirements and challenges of the HBA environment, and how we adapted the NetBSD kernel to deal with them. We have also provided results that show that NetBSD is a viable platform for these types of applications, but that more work is needed in order for it to reach its full potential.

We also believe that WEPE has the potential to enable NetBSD to be used in several applications that were previously off-limits, including system firmware and on MMU-less platforms.

7. Areas for future study

The semantics of Unix system calls make it difficult to write extremely high-performance applications. In particular, the implied ownership of a data buffer before, during, and after a system call is problematic. Furthermore, in the presence of a flat address space, there is no protection boundary to cross during a system call, complicating the use of existing zero-copy data movement schemes. Asynchronous APIs with explicit data ownership should be employed to address these issues.

The memory footprint is still somewhat larger than we would like for the HBA application. Some work should be done to identify dead code and eliminate it. Similarly, we should be able to improve the cache footprint of the software significantly in the kernel environment.

The extant profile analysis tools for Unix assume that time units are counted by the profiling subsystem [9]. This can result in confusing profile reports when using other types of profiling triggers, such as branch prediction misses, data cache misses, or TLB misses. Knowledge of other types of profile events should be added to these tools so that more meaningful reports can be generated.

Some performance gains could be achieved by offloading more processing onto hardware. For example, the Intel i82544 Gigabit Ethernet MAC can perform "TCP segmentation offload", which allows the host to provide a large data buffer, along with a template TCP header, to the MAC, which then performs the task of breaking up the large buffer into appropriately sized TCP segments. The NetBSD TCP/IP stack should be adapted to take advantage of such features.

The basic threading model in the kernel is different from that provided by **libpthread** in user space. Assumptions in one model or the other may adversely affect performance in the other space. It should be possible to either mitigate the adverse effects or develop tools to better measure the effects of scheduling on performance of an application.

8. References

[1] J. Satran et al. "iSCSI", Internet Draft draft-ietf-ips-iscsi-20.txt, pp. 107-108.

[2] J. Satran et al. "iSCSI", Internet Draft draft-ietf-ips-iscsi-20.txt, pp. 198-200.

[3] J. Hufferd. "iSCSI: The Universal Storage Connection", pp. 31-33. Addison-Wesley Publishing Company (2003)

[4] J. Hufferd. "iSCSI: The Universal Storage Connection", p. 25. Addison-Wesley Publishing Company (2003)

[5] "BusLogic Multi-Master Ultra SCSI Host Adapters for PCI Systems Technical Reference Manual", BusLogic, Inc. (1996)

[6] "Intelligent I/O (I2O) Architecture Specification, Version 2.0", I2O Special Interest Group (1999)

[7] RedBoot: *http://sources.redhat.com/redboot/*

[8] Iometer: *http://sourceforge.net/projects/iometer/*

[9] S. Graham, P. Kessler, M. McKusick. "gprof: A Call Graph Execution Profiler", proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6., pp. 120-126 (1982)

9. About the authors

Jason R. Thorpe is the Chief Science Officer of Wasabi Systems, Inc. A contributor to the NetBSD

Project since 1994, he has been involved in developing high-performance storage and networking systems using NetBSD for over eight years. Mr. Thorpe is a past participant in the IETF's TCP implementation working group, and also contributes to the GNU GCC, Binutils, and GDB projects.

Allen K. Briggs started working with BSD shortly after the release of 386BSD 0.0 by contributing to a BSD port to Apple's m68k-based Macintosh systems, which evolved into NetBSD/mac68k. Mr. Briggs has continued to work with NetBSD in his spare time, and in late 2000, joined Wasabi Systems, Inc., where he has been involved in a number of NetBSD-based embedded systems projects.